Capturing Register and Control Dependence in Memory Consistency Models with Applications to the Itanium Architecture

Lisa Higham[1], LillAnne Jackson[1,2], and Jalal Kawash[3,1]

[1] Department of Computer Science, The University of Calgary, Calgary, Canada
[2] Department of Computer Science, The University of Victoria, Victoria, Canada
[3] Department of Computer Science, American University of Sharjah, UAE
higham@cpsc.ucalgary.ca, jackson@cpsc.ucalgary.ca, jkawash@aus.edu

Abstract. A *complete* framework for modelling memory consistency that includes register and control dependencies is presented. It allows us to determine whether or not a given computation could have arisen from a given program running on a given multiprocessor architecture. The framework is used to provide an exact description of the computations of (a subset of) the Itanium instruction set on an Itanium multiprocessor architecture. We show that capturing register and control dependencies is crucial: a producer/consumer problem is solvable without using strong synchronization primitives on Itanium multiprocessors, but is impossible without exploiting these dependencies.

Keywords: Multiprocessor memory consistency, register and control dependency, Itanium, process coordination.

1 Introduction

To overcome inefficiency bottlenecks, modern shared memory multiprocessors have complicated memory organization such as replication in caches and write buffers, multiple buses, and out-of-order memory accesses. This causes processors to have differing views of the shared memory, which satisfy only some weak consistency guarantees. To program such systems, programmers need a precise specification of these guarantees, expressed as constraints on the outcomes of executions of programs. We provide a general and intuitive framework for defining the complete memory consistency model of a multiprocessor architecture including its control and register dependencies. The framework facilitates the specification of the exact set of computations that can arise from a multiprocessor system.

As a running example, we illustrate our framework with the Itanium programming language and architecture. Our techniques are exploited to specify the sets of computations that can arise from a multiprocessor program that uses a subset of the Intel Itanium instruction set when it is executed on an Itanium multiprocessor architecture. Since Itanium multiprocessors have a complicated architecture for which a complete correct description of its memory consistency, at the level of indivisible Itanium instructions, does not exist in the literature,

this is a second contribution of this paper. Another contribution is to show that including register and control dependencies in memory consistency models is crucial. We prove that a certain producer/consumer coordination problem is solvable without using strong synchronization primitives on Itanium multiprocessors, but is impossible without exploiting these dependencies.

Our complete framework is an extension of our previous work [5]. It maintains the ability to describe systems at different levels of abstraction and to prove their equivalence. But our previous framework could only capture constraints that arise from shared memory; it did not capture the constraints that arise from control dependencies in an individual processor's program, nor consistency constraints due to private register dependencies. While our previous framework could be used to determine if a given computation could arise from a given shared memory architecture, it did not extend to answering if the computation could arise from executing a given program on that architecture. These shortcoming are corrected in this paper. We have also used similar ideas to explore the possibility or impossibility of implementing some objects on various weak memory consistency machines [8–11]. A common feature of all of these is the notion of abstract specifications and implementations, as a generalization of the methodology used for algorithm design in linearizable [4] or sequentially consistent systems. Even though the systems we consider are substantially weaker than these, we can compose our implementations to achieve implementations of abstract objects on various multiprocessors with weak memory models (or to prove impossibilities).

Local dependencies are modelled in [1, 2]. Indeed, the framework of [1] contains many of the features of our framework, including the association of a program with an instantiation, which is similar to our computations, and the capturing of branching and register dependencies. We are unaware, however, of how to use that framework to describe systems at various levels and to prove equivalence.

There are several models for Itanium computations in the literature. Chatterjee and Gopalakrishnan [3] presented an operational model that they describe as a simple formal model for Itanium. Their model is simplified and does not include data dependencies. Yang, Gopalakrishnan, *et. al.* [18] specify Itanium memory ordering rules in terms that can be applied to verification using Prolog with a finite domain constant solver and a boolean Satisfiability checker. Joshi, Lamport, *et. al.* [16] applied the TLA+ specification language to the Itanium manual specifications [14] and use the TLC model checker on the resulting specifications. It was this work that brought about a clear description of the Itanium memory order [12], but it does not inclued a formal definition of local dependence order.

We use only enough of a simplified Itanium system to illustrate our framework with an Itanium running example. Other work [7] (also [15] in progress) focuses on the completed Itanium memory consistency. Thus we omit store release, load acquire, semaphore or memory fence instructions, and we do not consider procedure stack adjustments. Furthermore, we assume that memory

accesses that are not to identical locations do not overlap, all memory is in the same coherence domain, and all memory locations are cacheable (called WB in the Itanium manuals).

2 Modelling Multiprocessor Systems and their Computations

We model a multiprocessor system as a collection of *programs* operating on a collection of *objects* under some constraints called a *memory consistency model*.

Informally, as each processor of such a system executes its program, it issues a sequence of *operation invocations* on the objects of the system, and receives *operation responses* for each invocation. Matching each response with its invocation provides each processor with a sequence of *operations*, which is its "view" of the execution. We think of a *computation* as the collection of these views — one for each processor in the system. Because the response of an invocation is determined by the programs of the other processors, the asynchronous interaction of the processors, and the memory organization of the system, there are typically many computations possible for each system. For any such system we seek a way to determine exactly what computations are possible.

Although our framework is general, and can be used for any multiprocessor system, this paper focuses on the computations that can arise from a given program running on some typical multiprocessor architecture, say $\mathcal{A}$, where the objects are private registers and shared memory locations. We use a running example based on the Intel Itanium multiprocessor architecture to illustrate the definitions and their use.

Instructions and programs: The *programming language* of $\mathcal{A}$ consists of a collection of (machine specific) operations, called *instructions*, that perform a variety of load, store, arithmetic and logical functions. Instructions are partitioned into two classes:

- Branch instructions are all instructions that contain a conditional or unconditional branch to a specified label including procedure calls and procedure returns, and
- operational instructions are all other instructions.

Branch instructions transfer program control to a target instruction specified by a label, by operating on the value in the program counter register. Operational instructions move data from one register/memory location to another and/or perform arithmetic or logical functions on the data in registers/memory locations.[4]

Example - Itanium instruction: The Itanium operational instruction "ld8 $r2$=[$r1$]" takes the value from the memory location whose address is stored

[4] The instructions of a program are assumed to be in a format that has been prepared to be processed by an assembler, i.e., some instructions are preceded by labels and the target of a branch is specified by such a label.

in register $r1$, and places that value in register $r2$. In any execution, this instruction has associated with it some value v in $r1$, some value w in the memory location with address v and the value u stored in $r2$.

An *individual program* is a finite sequence of instructions from the programming language of $\mathcal{A}$.

Example - Itanium individual program: Figure 1 is an example of an Itanium individual program.

```
1.  loop:       ld8      r1 = [r3]        ;load r1 with data whose address is in r3
2.               add      r1 = #1, r1      ;add 1 to r1
3.               cmp.ge p1, p0=#3,r1 ;if 3 is greater than or equal to the value in r1 then
4.          (p1) br       loop             ; go back to loop
5.               st8      [r4], r1         ;store value of r1 into memory whose address is in r4
```

Fig. 1. An Itanium individual program

A *multiprogram* for an n processor machine with architecture $\mathcal{A}$ is a collection of n individual programs— one assigned to each processor.

Computations: An instruction that is augmented with *arbitrary* associated values (from the allowed domain), for all the registers and memory locations that it reads is a *completed instruction*. For a conditional branch instruction, "`br`(*cond*) label", the domain of register *cond* is $\{0, 1\}$, and associating one of these values resolves the conditional branch. So the completed form of this instruction is shortened to exactly one of two read operations that access *cond* and return 0 or 1 (namely, 0=read(*cond*) or 1=read(*cond*)). An *individual computation* of an individual processor is *any* sequence of completed instructions. We do not (yet) care what the instructions are or what values are associated with the instructions. A *(multiprocessor) computation* is a set of individual computations, one for each processor. Notice that a computation is defined non-operationally. It is not a single sequence describing an execution of the whole system, nor is there any relationship required between the register and memory values associated with the different completed instructions. Informally, a multiprocessor computation simply records the sequence of operational instructions performed (and therefore completed) by each processor and the values in condition registers that it read along the way.

Example - 2-processor computations: Three 2-processor computations are given in Figures 2, 3 and 4. (The actual values recorded in these computation can be bizarre, but they are still computations.)

p_1	p_2
ld8 $r1$ = $[r3]$ (ν_{r3}=1284, $\nu_{m(1284)}$=7)	ld8 $r1$ = $[r3]$ (ν_{r3}=1027, $\nu_{m(1027)}$=12)
add $r1$ = #1,$r1$ (ν_{r1}=15)	cmp.ge $p1$,$p0$ = #3,$r1$(ν_{r1}=4)
cmp.ge $p1$,$p0$ = #3,$r1$ (ν_{r1}=7)	add $r1$ = #1,$r1$ (ν_{r1}=6)
1=read($p1$)	0=read($p1$)
add $r1$ = #1,$r1$ (ν_{r1}=7)	st8 $[r4]$ = $r1$ (ν_{r4}=1104, ν_{r1}=7)
cmp.ge $p1$,$p0$ = #3, $r1$ (ν_{r1}=21)	
0=read($p1$)	
st8 $[r4]$ = $r1$ (ν_{r4}=101, ν_{r1}=17)	

Fig. 2. Computation 1 (arbitrary values are associated with instructions)

p_1	p_2
ld8 $r1$ = [$r3$] (ν_{r3}=1284, $\nu_{m(1284)}$=7)	ld8 $r1$ = [$r3$] (ν_{r3}=1027, $\nu_{m(1027)}$=12)
add $r1$ = #1,$r1$ (ν_{r1}=15)	add $r1$ = #1,$r1$ (ν_{r1}=4)
cmp.ge $p1$,$p0$ = #3,$r1$ (ν_{r1}=7)	cmp.ge $p1$,$p0$ = #3,$r1$(ν_{r1}=6)
1=read($p1$)	0=read($p1$)
add $r1$ = #1,$r1$ (ν_{r1}=7)	st8 [$r4$] = $r1$ (ν_{r4}=1104, ν_{r1}=7)
cmp.ge $p1$,$p0$ = #3, $r1$ (ν_{r1}=21)	
0=read($p1$)	
st8 [$r4$] = $r1$ (ν_{r4}=101, ν_{r1}=17)	

Fig. 3. Computation 2

p_1	p_2
ld8 $r1$ = [$r3$] (ν_{r3}=1284, $\nu_{m(1284)}$=2)	ld8 $r1$ = [$r3$] (ν_{r3}=1280, $\nu_{m(1280)}$=5)
add $r1$ = #1,$r1$ (ν_{r1}=2)	add $r1$ = #1,$r1$ (ν_{r1}=5)
cmp.ge $p1$,$p0$ = #3,$r1$ (ν_{r1}=3)	cmp.ge $p1$,$p0$ = #3,$r1$(ν_{r1}=6)
1=read($p1$)	0=read($p1$)
add $r1$ = #1,$r1$ (ν_{r1}=3)	st8 [$r4$] = $r1$ (ν_{r4}=1284, ν_{r1}=0)
cmp.ge $p1$,$p0$ = #3, $r1$ (ν_{r1}=4)	
0=read($p1$)	
st8 [$r4$] = $r1$ (ν_{r4}=1280, ν_{r1}=1)	

Fig. 4. Computation 3

Central goal: We aim to construct a comprehensive framework that is capable of capturing the control and register dependences of any multiprocessor architecture $\mathcal{A}$, and the constraints that arise from the memory organization of $\mathcal{A}$. The definition of any such $\mathcal{A}$ must be precise enough to decide the following *Architecture-Computation* decision problem.

Input: a multiprogram P, containing an individual program for each processor of $\mathcal{A}$, and a multiprocessor computation C.

Question: Could C arise from executing P on $\mathcal{A}$?

Example - An instance of the Architecture-Computation Problem: Let $IP = \{prog_1, prog_2\}$ where $prog_1$ and $prog_2$ are both the program in Figure 1. When applied to our running example, the Architecture-Computation problem becomes: "Could Computation 1 (respectively, 2 or 3) in Figure 2 (respectively, 3 or 4) arise from running IP on an Itanium multiprocessor?"

Two major steps are required to answer the Architecture-Computation question. The first is to determine if each processor's individual computation (ignoring the associated values) could have arisen from its program. If the answer is yes, then the second step is to determine whether or not the individual program sequences could have interacted in such a way under architecture $\mathcal{A}$ as to produce the values associated with each of the instructions in the computation.

Step 1: Program graphs and computational forms. To answer the first question, each processor's program is modelled as a directed graph, such that any computation of an individual program must correspond to some path through its graph. Specifically, for any individual program *prog* associate a directed node-labeled graph called the *program-graph of* prog, denoted $\mathcal{G}(prog)$, as follows.

Nodes of $\mathcal{G}(prog)$ For every operational instruction `inst`, there is a vertex η(`inst`) with label `inst`. For every conditional branch instruction `br` of the form "br(*cond*) label:", there are two vertices, η_0(`br`) with label 0=read(*cond*)

and η_1(`br`) with label 1=read(*cond*). Henceforth, the labels of vertices of $\mathcal{G}(prog)$ are referred to as *node-labels* to distinguish them from the labels of instructions in *prog*. Notice that unconditional branches do not correspond to a node.

Directed edges of $\mathcal{G}$(*prog*) For any instruction `inst` in *prog*, associate a set of vertices, vertices(`inst`), by:

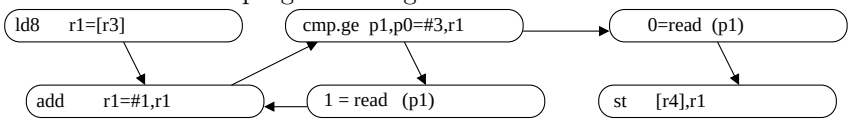
- for an operational instruction `inst`, vertices(`inst`) = $\{\eta$(`inst`)$\}$,
- for a conditional branch instruction `br`, vertices(`br`) = $\{\eta_0$(`br`), η_1(`br`)$\}$,
- for an unconditional branch instruction, say "br continue-here", then vertices($brcontinue-here$) = vertices($\widehat{\texttt{inst}}$) where $\widehat{\texttt{inst}}$ is the instruction with label "continue-here:".

Denote by `p-succ(inst)` the instruction in *prog* that follows `inst`. For each vertex η in $\mathcal{G}(prog)$, define `g-succ(`η`)` by:

- for an operational instruction, `inst`,
  `g-succ(`η`(inst))` = vertices(`p-succ(inst)`)
- for a conditional branch instruction, `br`= br(*cond*) label", where $\widehat{\texttt{inst}}$ is the instruction with label "label:", `g-succ(`η_0`(br))` = vertices(`p-succ(br)`) and `g-succ(`η_1`(br))` = vertices($\widehat{\texttt{inst}}$).

Finally, for each vertex η in $\mathcal{G}(prog)$, there is a directed edge from η to every vertex in `g-succ(`η`)`.

Example - Program graph for an Itanium program: The program graph of the Itanium individual program of Figure 1 is:



Let `start` be the first instruction in an individual program *prog*. An *individual program sequence for* prog is the sequence of the node-labels on any (possibly non-simple) directed path in $\mathcal{G}(prog)$ that begins with any vertex in vertices(`start`). Let $P = \{prog_1, prog_2, ..., prog_n\}$ be a multiprogram. Then $CF = \{$*prog-seq*$_1$, *prog-seq*$_2$, ..., *prog-seq*$_n\}$ is a *computational form of* P if and only if for $1 \leq i \leq n$, *prog-seq*$_i$ is an individual program sequence for $prog_i$. A computation C *agrees with* CF if, for each i, the ith sequence of instructions in C, (ignoring values) is the same as *prog-seq*$_i$. From these definitions we have:

Claim. A multiprocessor computation can be a computation of a multiprogram P only if it agrees with a computational form of P.

Example - Computational form for an Itanium 2-processor multiprogram: $CF = \{$*prog-seq*$_1$, *prog-seq*$_2\}$, where *prog-seq*$_1$ and *prog-seq*$_2$ are given in Figure 5, is a computational form of the Itanium multiprogram IP in the running example.

$prog\text{-}seq_1$		$prog\text{-}seq_2$	
ld8	$r1 = [r3]$	ld8	$r1 = [r3]$
add	$r1 = \#1,r1$	add	$r1 = \#1,r1$
cmp.ge	$p1,p0 = \#3,r1$	cmp.ge	$p1,p0 = \#3,r1$
1=read($p1$)		0=read($p1$)	
add	$r1 = \#1,r1$	st8	$[r4] = r1$
cmp.ge	$p1,p0 = \#3,r1$		
0=read($p1$)			
st8	$[r4] = r1$		

Fig. 5. A computational form

Notice that a computational form contains only operational instructions; there are no branch instructions. In the program graph each conditional branch instruction is replaced by one node for each of the two possible outcomes. The outgoing edges of these nodes lead to the next instruction that is correct given the value of the condition. Each unconditional branch is replaced only by edges.
Example - Agreement of computations with an Itanium computational form: Both computations in Figures 3 and 4 agree with the computational form for IP in Figure 5 but the computation in Figure 2 does not because the cmp.ge and add instructions of p_2 are in the opposite order from the order of the add and cmp.ge node-labels in the program graph.

Step 2: From computational forms to computations: To answer the second question, the rules of interaction for the machine with architecture $\mathcal{A}$ are modelled as a collection of constraints on various subsets of the instructions of a computation. For each subset there is a specified partial order, for which there must be a total order extension that is *valid* (to be defined next). Furthermore there may be agreement properties required between some of the total order extensions.

Validity

Meaning of instructions: To define validity, each entry in a computational form (i.e. each node-label, ηl) is assigned a *meaning*, denoted $\mathcal{M}(\eta l)$, by mapping it to a short program that uses only read, write, arithmetic and logic operations on variables. Specifically, the programming language `Trivial`:

- has two kinds of objects: single-reader/single-writer atomic variables and multi-reader/multi-writer atomic variables, and
- supports five types of operations: read and write operations on atomic variables, arithmetic and logic operations on single-reader/single-writer atomic variables, and "if-conditional-then-operations-else-operations"

Recall that each node-label, ηl, is either an operational instruction or 0=read($cond$), or 1=read($cond$). For $\eta l \in \{0 = \text{read}(cond), 1 = \text{read}(cond)\}$ define $\mathcal{M}(\eta l)$ to be the identity function. If ηl is an operational instruction then $\mathcal{M}(\eta l)$ is defined to be the short `Trivial` program that re-expresses the semantics of the instruction as specified by the instruction manual of $\mathcal{A}$.
Example - Meaning of some Itanium operational instructions: Figure 6 gives four examples. These can be extracted from the Itanium Instruction Set Reference manual [13].

$\mathcal{M}$(*ld8 r1=⌈r3⌋*)	$\mathcal{M}$(*add r1=#1,⌈r3⌋*)	$\mathcal{M}$(*cmp.eq p1,p2=r1,r3*)	$\mathcal{M}$(*st8 ⌈r4⌋=r5*)
$\nu_1 = \text{read}(r3)$ $\nu_2 = \text{read}(\nu_1)$ $\text{write}(r1, \nu_2)$	$\nu_1 = \text{read}(r3)$ $\nu_2 = \text{read}(\nu_1)$ $\nu_3 = 1 + \nu_2$ $\text{write}(r1, \nu_3)$	$\nu_1 = \text{read}(r2)$ $\nu_2 = \text{read}(r3)$ if ($\nu_1 == \nu_2$) then { $\text{write}(p2, 1)$ $\text{write}(p1, 0)$ } else { $\text{write}(p2, 0)$ $\text{write}(p1, 1)$ }	$\nu_1 = \text{read}(r5)$ $\nu_2 = \text{read}(r4)$ $\text{write}(mem(\nu_2), \nu_1)$
(a)	(b)	(c)	(d)

Fig. 6. Meaning of four Itanium instructions

Recall that a completed instruction, `inst`, has values associated with each register or shared memory location that it reads, and the variables in $\mathcal{M}$(`inst`) are assigned by reading, writing, or performing arithmetic/logic operations on the values in these registers. If the register values in the completed instruction `inst` are used to compute the values of the corresponding variables in $\mathcal{M}$(`inst`), the resulting sequence of `Trivial` operations is called the *derived completed meaning of* `inst`, denoted $\mathcal{DM}$(`inst`). Because the operations are completed, the conditional operations in this sequence can be resolved, so the derived completed meaning of `inst` is reduced to a sequence I of completed read, write and arithmetic/logic operations. Define the *remote derived completed meaning of* `inst`, denoted $\mathcal{RDM}$(`inst`), to be the subsequence of I consisting of only the completed write operations to shared memory locations.

Example - Derived completed meanings: The derived completed meaning of the completed instruction ld8 $r1$ = [$r3$] (ν_{r3}=1284, $\nu_{m(1284)}$=7) is "1284 = read($r3$), 7 = read(1284), write($r1, 7$)" whereas its remote derived completed meaning is the empty list. For st8 [$r4$] = $r5$ (ν_{r4}=1280, ν_{r5}=1) the derived completed meaning is 1 = read($r5$), 1280 = read($r4$), write($mem(1280), 1$) and the remote derived meaning is just write($mem(1280), 1$).

Definition of validity of a sequence of completed instructions: Let $S = s_1, s_2, \ldots, s_k$ be any sequence of completed instructions of a computation. The *computed meaning of* S is the sequence formed by concatenating $\mathcal{DM}(s_i)$ for i from 1 to k. For a given individual program p, the *computed meaning of* S *for* p is the sequence formed by concatenating as follows. Replace each s_i that is an instruction by p with $\mathcal{DM}(s_i)$; replace each s_i that is an instruction by a processor different from p with $\mathcal{RDM}(s_i)$. Notice that both the computed meaning of S and the computed meaning of S for p are sequences of read, write and arithmetic/logic operations on atomic variables, so it is straightforward to determine if such a sequence is valid. Specifically, it is *valid* if each read returns the value of the last write to the same variable and all arithmetic/logic operations on variables are correct. Finally, a sequence S of completed instructions from a computation is *valid (respectively, valid for p)* if the computed meaning of S (respectively, the computed meaning of S for p) is valid.

Partial orders and memory consistency models The final task is to capture the rules that govern executions of multiprograms under architecture $\mathcal{A}$ as defining properties of the computations that can be produced. These properties, collectively called a memory consistency model, are different for every multiprocessor machine, but have the same general structure. They are expressed as a

collection of partial orders relations on the completed instructions of a computation. Each of these partial orders is required to have a total order extension that

- is valid and
- shares some agreement properties with other partial orders.

Example - The Itanium memory consistency model[5]: Let $I(\mathcal{C})$ be the set of all completed instructions in a computation C. $I(\mathcal{C})|p$ denotes the subset of $I(\mathcal{C})$ in processor p's program sequence; $I(\mathcal{C})|x$ denotes the subset whose meaning contains a read or write operation on (register or shared memory) variable x; and $I(\mathcal{C})|br$ is the subset whose meaning is 0=read(*cond*) or 1=read(*cond*), where cond is a (condition) register. $I(\mathcal{C})|r$ denotes the subset containing only the instruction instances that contain a read operation on a shared memory variable; $I(\mathcal{C})|w$ the subset containing only the instruction instances that contain a write operation on a shared memory variable. The relation $(I(\mathcal{C}), \xrightarrow{prog})$, called *program order*, is the set of all pairs (i, j) of completed instructions that are in the same individual computation of C and such that i precedes j in that sequence. For any partial order relation $(I(\mathcal{C}), \xrightarrow{y})$, the notation $i \xrightarrow{y} j$ is used interchangeably with $(i, j) \in (I(\mathcal{C}), \xrightarrow{y})$.

Define the following partial orders:

- Local dependence order $(I(\mathcal{C}), \xrightarrow{dep_p})$: For $i, j \in I(\mathcal{C})|p$, $i \xrightarrow{dep_p} j$ if $i \xrightarrow{prog} j$ and either
  Register: $i, j \in I(\mathcal{C})|x$, where x is a register, or
  Branch: $i \in I(\mathcal{C})|br$.
- Orderable order $(I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{ord_p})$ for each $p \in P$: $i \xrightarrow{ord_p} j$ if $i, j \in I(\mathcal{C})|p \cup I(\mathcal{C})|w$ and $i \xrightarrow{prog} j$ and $i, j \in I(\mathcal{C})|x$ and ($i \in I(\mathcal{C})|w$ or $j \in I(\mathcal{C})|w$)

Itanium memory consistency definition: A computation C satisfies Itanium consistency if for each $p \in P$, there is a total order $\xrightarrow{S_p}$ of the operations $I(\mathcal{C})|p \cup I(\mathcal{C})|w$ that is valid for p, such that

1. $(I(\mathcal{C})|p, \xrightarrow{dep_p}) \subseteq (I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{S_p})$, (Local requirement) and
2. $(I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{ord_p}) \subseteq (I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{S_p})$, (Orderable requirement) and
3. If $i_1, i_2 \in I(\mathcal{C})|x|w$ and $i_1 \xrightarrow{S_p} i_2$ then $i_1 \xrightarrow{S_q} i_2$, $\forall q \in P$, (Same Memory agreement) and
4. There does not exist a cycle of $i_1, i_2 \ldots i_k \in I(\mathcal{C})|w$ where $i_j \in I(\mathcal{C})|p_j, \forall j \in \{1, 2, \ldots k\}$ and $k \leq n$ such that: $i_k \xrightarrow{S_1} i_1$, and $i_1 \xrightarrow{S_2} i_2$, and $i_2 \xrightarrow{S_3} i_3$ ... and $i_{k-1} \xrightarrow{S_k} i_k$ (Cycle-free agreement)

Define Itanium-Dep (Itanium minus dependence) to be identical to the preceding definition, without the Local requirement.

Example - Itanium-Dep and Itanium consistency: Computation 3 of Figure 4 satisfies Itanium-Dep. The following sequences satisfy the Orderable requirement, the Same Memory agreement, and Cycle-free agreement properties.

[5] Itanium provides a rich instruction set, which includes semaphore and fence instructions. The definition formulated here ignores acquire, release, and fence instructions. The development and proofs for the general Itanium definition are elsewhere [7].

show that Computation 3 satisfies Itanium-Dep requirements and agreement properties:

S_{p_1} : ld8 $r1$ = [$r3$] (ν_{r3}=1284, $\nu_{m(1284)}$=2) $\xrightarrow{S_{p_1}}$ st8$_{p_2}$ [$r4_{p_2}$] = $r1_{p_2}$ ($\nu_{r4_{p_2}}$=1284, $\nu_{r1_{p_2}}$=0) $\xrightarrow{S_{p_1}}$ st8 [$r4$] = $r1$ (ν_{r4}=1280, ν_{r1}=2) $\xrightarrow{S_{p_1}}$ add $r1$ = #1,$r1$ (ν_{r1}=2) $\xrightarrow{S_{p_1}}$ cmp.ge $p1$,$p0$ = #3,$r1$ (ν_{r1}=3) $\xrightarrow{S_{p_1}}$ 1=read($p1$) $\xrightarrow{S_{p_1}}$ add $r1$ = #1,$r1$ (ν_{r1}=3) $\xrightarrow{S_{p_1}}$ cmp.ge $p1$,$p0$ = #3, $r1$ (ν_{r1}=4) $\xrightarrow{S_{p_1}}$ 0=read($p1$)

S_{p_2} : st8 [$r4$] = $r1$ (ν_{r4}=1284, ν_{r1}=0, $\nu_{m(1284)}\leftarrow$0) $\xrightarrow{S_{p_2}}$ ld8 $r1$ = [$r3$] (ν_{r3}=1280, $\nu_{m(1280)}$=5, $\nu_{r1}\leftarrow$5) $\xrightarrow{S_{p_2}}$ add $r1$ = #1,$r1$ (ν_{r1}=5 $\nu_{r1}\leftarrow$6) $\xrightarrow{S_{p_2}}$ cmp.ge $p1$,$p0$ = #3, $r1$(ν_{r1}=6, $\nu_{p1}\leftarrow$0) $\xrightarrow{S_{p_2}}$ 0=read($p1$) $\xrightarrow{S_{p_2}}$ st8$_{p_1}$ [$r4_{p_1}$] = $r1_{p_1}$ (ν_{r4}=1280, $\nu_{r1_{p_1}}$= 1, $\nu_{m(1280)}\leftarrow$1)

To show that these sequences are valid we provide `Trivial` instructions for the derived completed meaning or remote derived completed meaning for each instruction in the sequence above. Square brackets, [ ], delineate the operations for each instruction.

Validity Sequence for S_{p_1}: [1284 = read($r3$), 2 = read(1284), write($r1$, 2)] [write(1284, 0)] [2 = read($r1$), 1280 = read($r4$), write(1280, 2)] [2 = read($r1$), 3 = 1 + 2, write($r1$, 3)] [3 = read($r1$), if (#3 $geq > 3$) then write($p1$, 1) else write($p1$, 0)] [1=read($p1$)] [3 = read($r1$), 4 = 1 + 3, write($r1$, 4)] [4 = read($r1$), if (#3 $geq > 4$) then write($p1$, 1) else write($p1$, 0)] [0=read($p1$)]

Validity Sequence for S_{p_2}: [0 = read($r1$), 1284 = read($r4$), write(1284, 0)] [1280 = read($r3$), 5 = read(1280), write($r1$, 5)] [5 = read($r1$), 6 = 1 + 5, write($r1$, 6)] [6 = read($r1$), if (#3 geq 6) then write($p1$, 1) else write($p1$, 0)] [0=read($p1$)] [write(1280, 1)]

However, Computation 3 does not satisfy Itanium because it must extend Local dependence order, which requires that the st8 instruction in each sequence follows the add instructions in that sequence, which would break validity. A computation that is similar to Computation 3 except that the st8 instruction by p_1 has values (ν_{r4}=1280, ν_{r1}=3) and the st8 instruction by p_2 has values (ν_{r4}=1280, ν_{r1}=6) satisfies Itanium.

3 Consequences of Ignoring Dependence Order

Ignoring register and control dependences can lead to erroneous conclusions about the capabilities or limitations of the architecture under consideration. This section illustrates such an error using a simple producer-consumer example and the Itanium architecture.

3.1 A simple producer-consumer multiprocessor system

Informally, the simple producer-consumer (SPC) multiprocessor system defined here has one producer program, producing items, which are consumed by a single consumer program. The producer and consumer take turns in producing and

consuming items. This specialized system can be defined using the framework of Section 2 as follows:[6]

Objects: A *SPC object* X supports two operations P-write(X,ι) and ι=C-read(X). The semantics of these operations are exactly the same as the read and write operations on an atomic variable. However, the validity requirement is substantially stronger.

Validity: A sequence of (C-read and P-write) operations on SPC object X is valid if: (1) it starts with a P-write operation, (2) alternates between C-read and P-write operations, (3) each C-read returns the value written by the immediately preceding P-write operation, and (4) it has an equal number of P-write and C-read operations.[7]

Programs: There are two programs: the producer repeatedly 'P-writes' the SPC object, and the consumer repeatedly 'C-reads' it, for a specified (possibly infinite) number of rounds.

SPC System: The SPC multiprocessor system consists of one SPC object X and a multiprogram of one producer and one consumer. The memory consistency model is sequential consistency[17].

3.2 Proving relationships between multiprocessor systems

Given a multiprocessor system with multiprogram P, objects J, and a memory consistency model M, we call (P, J) a *multiprocess* and when J consists entirely of atomic variables and registers, we call (J, M) an M *platform*.

The *specified* SPC multiprocess (P, J) gives rise to a set of allowable (specification) computations, $\mathcal{C}$ in Figure 7, under sequential consistency. We model the execution of the *specified* multiprocess on the *target* Itanium architecture, as a transformation, τ. This transformation replaces the specified objects, operations, and memory consistency model respectively with target (Itanium) objects, instructions, and memory consistency model. Specifically, an operation on a specified object is *transformed* to the target objects by providing a subroutine for the operation's invocation where this subroutine uses only instruction invocations on the target objects, $\widehat{J}$. If the specified operation returns an output, then the subroutine must return a value of the same type as this output. An object in J is *transformed* to the target object(s) in $\widehat{J}$ by transforming each of its operations. A transformation of each of the objects of a specified multiprocess to objects of the target multiprocess can be naturally extended to a *program transformation* by

[6] A general producer-consumer system definition may allow several producers and consumers and more complex shared objects such as queues, where producers enqueue items and consumers dequeue them. This very restricted definition (two programs and a queue of size one) suffices for this section. Furthermore, without exploiting additional Itanium synchronization mechanisms, such as fences, acquires, or releases, a solution for the general case is not possible.

[7] Note that strict alternation between P-write and C-read operations is not necessary at lower levels. This specification-level requirement can be satisfied at an implementation-level as long as operations appear as if they do alternate.

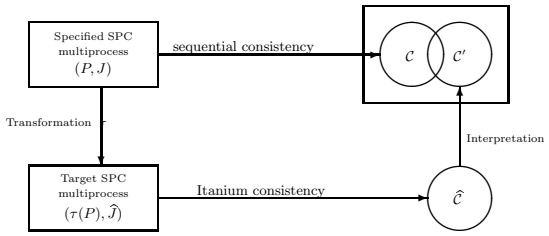


Fig. 7. Program transformation and computation interpretation.
If $\mathcal{C}' \subseteq \mathcal{C}$, then the transformation is an implementation.

replacing each operation invocation in the specified multiprogram with the subroutine for that operation invocation. The transformed multiprogram together with the target objects $(\tau(P), \widehat{J})$ comprise the *transformed multiprocess*.

Hence, each P-write and C-read operations will be transformed to subroutines that only make use of Itanium instructions. The transformed multiprocess gives rise under Itanium consistency to an allowable set of computations, $\widehat{\mathcal{C}}$. Any computation in $\widehat{\mathcal{C}}$ can be *interpreted* as a computation of the specified multiprogram by attaching to each operation invocation of the specified multiprogram the value returned by the corresponding subroutine. Thus, the set $\hat{C}$ of computations of the target system provides a set of interpreted computations $\mathcal{C}'$ of the specified multiprocess.

A transformation of a specified multiprocess is called an *implementation* of the specified system, if, informally, (in Figure 7) the set $\mathcal{C}'$ of computations produced by traveling the long way around is a non-empty subset of the computations $\mathcal{C}$ allowed by the specified multiprocessor system. The formal details, including stronger (than 'implementations') relations between systems, are elsewhere [5].

3.3 In the presence of dependence

Use p and c to refer to the producer and consumer programs, respectively. For the transformation of the SPC multiprocess given in Figure 8, consider the last load in two consecutive iterations of p. We use dependence order and validity to prove that at least one copy of c's store at line 10 occurs in p's sequence $(I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{S_p})$ between these two loads. Also consider the last load in two consecutive iterations of c. We use dependence order and validity to prove that at least one copy of p's store at line 4 occurs in c's sequence $(I(\mathcal{C})|c \cup I(\mathcal{C})|w, \xrightarrow{S_c})$ between these loads. Orderable order and the same memory agreement property force the stores to strictly alternate in each sequence, ensuring that the c loads every produced value. The proof of the following theorem is long and tedious and is elsewhere [6].

Produce

Pseudo Code	$\tau((P{-}write(\mathrm{X}, item))$:		
	0	mov r2=Q	; r2 contains the address of Q
while (Q $\neq$ 0) wait;	1 L:	ld8 r1=[r2]	; load r1 with Q
	2	cmp.neq p1,p0 = r1,r0	; is r1 == 0?
	3	(p1) br L:	; If not, go back to L
Q $\longleftarrow$ item	4	st8 [r2]= item	; store item into Q

Consume

Pseudo Code	$\tau((C - read(\mathrm{X}))$ returns item in $r3$:		
	5	mov r2=Q	; r2 contains the address of Q
while (Q = 0) wait;	6 M:	ld8 r1=[r2]	; load r1 with Q
	7	cmp.eq p1,p0 = r1, r0	; is r1 == 0?
	8	(p1) br M:	; If so, go back to M
item $\longleftarrow$ Q	9	ld8 r1=[r2]	; load r1 with Q
Q $\longleftarrow$ 0	10	st8 [r2]=r0	; store 0 into Q
	11	mov r3, r1	; put consumed item into r3

Fig. 8. A transformation of the specified SPC object, X, on Itanium, using the target objects $\{Q, r1_p, r2_p, p1_p, r1_c, r2_c, r3_c, p1_c\}$

Theorem 1. *Transformation τ in Figure 8 is an implementation of the SPC system on an Itanium platform.*

3.4 In the absence of dependence

While the SPC system has an implementation on an Itanium platform, we show in this section that such an implementation does not exist on an Itanium-Dep platform.

First of all, we show how any transformation respecting the pseudo code in Figure 8 fails under only Itanium-Dep. Specifically, in the C-read pseudo code, the Orderable order does not always require program order between two load instructions, even if they access the same atomic variable (Q in this case). The consumption load (load of Q in 'item $\leftarrow$ Q') may be ordered in S_c before any load instruction resulting from the spin-loop, including the load that writes a value that consequently ends the loop. This load signals to the consumer to proceed with consumption. However under Itanium-Dep, the consumer can proceed to consumption before even performing any loads or checking the condition in the spin-loop. That is, a consumer may consume a never-produced or a previously consumed item. This 'early' load decides the value to be returned in the interpretation by the corresponding C-read. Any attempt to construct a valid total order for the producer-consumer object X will fail. The failure of this algorithm is general as we argue next.

Theorem 2. *There does not exist an implementation of the SPC system on an Itanium-Dep platform.*

Proof: Assume there exists an implementation β of the SPC system on an Itanium-Dep platform. Then the interpretation of any computation of the transformed SPC multiprocess will be valid for the SPC object X. By this validity requirement, the β(C-read(X)) subroutine does not return before it finds a new produced and never consumed before item. Hence, c must *wait* in β(C-read(X))

until a new produced item is available for consumption. In Itanium-Dep, the only way by which c waits for p is through the use of a spin-loop. The spin-loop has at least one load instruction, *spin-load*.

Furthermore, there has to be a load instruction after which the value to be returned by β(C-read(X)) will be decided and does not change until β(C-read(X)) returns. Call this load the *consumption load*. Obviously, the consumption load must follow in program order some of the spin-loads. We argue that Itanium-Dep can give rise to a computation that behaves as if the consumption load is completed before the spin-loop starts.

If between a spin-load and the following (in program order) consumption load there are any stores, then these stores cannot be to the same variable that the consumption load is accessing, otherwise the communicated produced item may be changed by the consumer and lost. A store that deterministically leaves the value of an atomic variable, say v, unchanged does not necessarily write the same value back to v in Itanium-Dep. Writing the same value stored in v back to v (something of the form: $v \leftarrow v$) requires a load that precedes in program order the store. What we have now in $prog\text{-}seq_{\beta(c)}$ is:

ld8 r_1=[r] ; begin spin-loads, register r has the address of v
ld8 r_2=[r]
...
ld8 r_k=[r] ; another spin-load (not necessarily the last of the loads in the loop)

ld8 r_{k+1}=[r] ; load resulting from writing v to v
st8 [r]=r_{k+1} ; writing v back to v
...
ld8 r_{k+i}=[r]; the consumption load

Itanium-Dep is insufficient to enforce any ordering between ld8 r_{k+1}=[r] and any of the k spin-loads in $(I(\mathcal{C})|c \cup I(\mathcal{C})|w, \xrightarrow{S_c})$. Hence, it is always possible for r_{k+1} to be assigned a never-produced (the item is not the result of β(P-write(X, ι))) item or an item that has already been consumed.

Finally, any store to a different variable between (in program order) the k spin-loads and the consumption load cannot restore the lost program order in S_c between the k spin-loads and the consumption load. That is, consumption can happen earlier than it should be, and we can end up with an invalid sequence of P-write and C-read operations. Hence, the transformation β cannot be an implementation, since in Figure 8, there is at least one computation in $\mathcal{C}'$ that cannot be in $\mathcal{C}$. ■

References

1. A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. on Parallel and Distributed Systems*, 14(5):502–515, 2003.

2. H. Attiya and R. Friedman. Programming DEC-Alpha based multiprocessors the easy way. In *Proc. 6th Int'l Symp. on Parallel Algorithms and Architectures*, pages 157–166, 1994. Technical Report LPCR 9411, Computer Science Department, Technion.
3. P. Chatterjee and G. Gopalakrishnan. Towards a formal model of shared memory consistency for intel itaniumtm. In *Proc. 2001 IEEE International Conference on Computer Design (ICCD)*, pages 515–518, Sept 2001.
4. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.
5. L. Higham, L. Jackson, and J. Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM Trans. on Computer Systems*. In Press.
6. L. Higham, L. Jackson, and J. Kawash. Capturing register and control dependence in memory consistency models with applications to the Itanium architecture. Technical Report 2006-837-30, Department of Computer Science, The University of Calgary, July 2006.
7. L. Higham, L. Jackson, and J. Kawash. Programmer-centric conditions for Itanium memory consistency. Technical Report 2006-838-31, Department of Computer Science, The University of Calgary, July 2006.
8. L. Higham and J. Kawash. Tight bounds for critical sections on processor consistent platforms. *IEEE Trans. on Parallel and Distributed Systems*. In Press.
9. L. Higham and J. Kawash. Critical sections and producer/consumer queues in weak memory systems. In *Proc. 1997 Int'l Symp. on Parallel Architectures, Algorithms, and Networks*, pages 56–63, December 1997.
10. L. Higham and J. Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. 12th Int'l Symp. on Distributed Computing, Lecture Notes in Computer Science volume 1499*, pages 201–215, September 1998.
11. L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *Proc. of the 7th Int'l Conf. on High Performance Computing, Lecture Notes in Computer Science volume 1970*, pages 355–366, December 2000.
12. Intel Corporation. A formal specification of the intel itanium processor family memory ordering. http://www.intel.com/, Oct 2002.
13. Intel Corporation. *Intel Architecture Architecture Software Developer's Manual: Volume 3: Instruction Set Reference*. Intel Corporation, Oct. 2002.
14. Intel Corporation. Intel itanium architecture software developer's manual, volume 2: System architecture. http://www.intel.com/, Oct 2002.
15. L. Jackson. Complete framework for memory consistency with applications to Itanium multiprocessors. Ph.D. Dissertation, in preperation.
16. R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla, 2003.
17. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.
18. Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Proc. 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2003.