Project Report for CSE548

The Study on the Optimality of Tomasulo's Algorithm

Xin(Luna), Dong

Gang, Zhao

March 12th, 2002

Department of Computer Science and Engineering, University of Washington

Project Report for CSE548

The Study on the Optimality of Tomasulo's Algorithm

Abstract

In this report we investigate the optimality of Tomasulo's Algorithm, the widely used dynamic scheduling algorithm in last thirty years. We firstly model a generic data-driven system as the reference system, and an ideal initial model for Tomasulo's Algorithm as the starting point of our discussion. Then a series of models with decreasing assumptions are built. The assumptions include bus delay and bandwidth, instruction window size, functional units number, reservation station size, and instruction type and the combination impacts of these factors. For each model, we either theoretically prove the optimality or find counterexamples. We show that Tomasulo's Algorithm is optimal when there are infinite hardware resources and no Load/Store instructions. But it cannot win under the two limitations, as the underlying problem is essentially a NP-hard problem. Some improvements are proposed to boost the scheduling performance from the original algorithm.

Key words: Tomasulo's algorithm, optimality, out-of-order execution, dynamic scheduling

Index

1. Introduction

Tomasulo's Algorithm provides a way to effectively exploit parallelism on instruction level and enhance instruction throughput. It is a hardware algorithm to remove name dependences while minimizing stalls caused by data dependences. [1] Due to its significant efficiency, Tomasulo's Algorithm has become an integral part of today's RISC processing cores. Moreover, the basic ideas of the algorithm, out-of-order execution and register renaming, are widely used in the state-of-art microprocessor architecture design.

As the need for peak-throughput has grown and the cost for hardware complexity has shrunk, it seems we have good reasons to aggressively pursue better algorithm for improving performance. The natural questions to ask are – whether Tomasulo's Algorithm is optimal among many dynamic scheduling algorithms, whether there is room for improvement, and what should be a wise trade-off between time and cost. By optimality, we mean the generated instruction execution stream needs the least execution clock cycles.

To give out the formal analysis of the performance of Tomasulo's Algorithm, a set of mathematical models are built up in this paper. The key features of this approach are:

- Optimality is proved by comparison with a reference system, which can produce all the possible instruction execution orders without violating data dependences.
- Time description variables are led into both systems to record the exact finish time of every instruction. On the other hand, as our interest lies in instruction flow but not correctness, no variables are used to record computation results.
- A group of ideal assumptions are made at first, and dropped one by one. In this way, we can figure out the crucial factors to maintain the algorithm's optimality.

By comparing the two systems in each model, we have such observations:

- When we have unlimited hardware resources, and no hardware except the functional units causes delay, Tomasulo's Algorithm is optimal. This is proved formally.
- When transmission delay is brought into the model, above conclusion remains.
- When the instruction window is limited, the algorithm is still optimal.
- When other hardware, including functional units and bus bandwidth, is restricted, the optimality does not hold any more. Improvement can be made to give wiser scheduling, which can generally produces better execution streams, but at the same time brings high overhead and complexity.
- When the number of instructions we can issue per cycle is restricted, which means the hardware cannot look ahead infinitely, the algorithm is not optimal and no improvement technique to the algorithm itself can help.
- When instructions of Load/Store are taken into account, the algorithm is not optimal and can be improved using the mechanism similar to RS.

1.1 Related Work

As the need for advanced validation techniques develops, a lot of efforts are made on proving the correctness of Tomasulo's Algorithm. Among them, [2,3] base their formal correctness proof on refinement. Recent papers [4,5,6,7] use model-checking to verify the algorithm mechanically with the help of verifiers. Our mathematical models are inspired by [2]. This paper introduces an intermediate model, which in later work is regarded as redundant and can be simplified. However, it is just this model that forms the reference system in our optimality proof. We base our work on the models in [2] and add time recording variables to make comparison between the time needed to execute the same program in either system.

Compared to formal validations of Tomasulo's Algorithm, there has been relatively little theoretical analysis of its performance in the literature to date. [8] examines different scheduling techniques, most of which are implementation of Tomasulo's Algorithm. [9] checks the impact brought by different ways to organize reservation station buffers. However, both of the papers focus on the implementation details of the algorithm without saying anything about its own effectiveness. Also, they both make investigations by simulation. On the contrary, we based our proof and analysis on mathematical models. At the end of the report, we made a conclusion at a generic level.

1.2 Paper Organization

This report is structured as following. Before the discussion, we first clarify some assumptions for all models. We make ideal assumptions and builds up a model under these unrealistic assumptions in section 2. After that, two of the assumptions are removed one by one to make the model more down-to-earth. Formal proofs of optimality are given for each of the models. In section 6, the critical assumption of unlimited hardware resources are taken off and counter-examples are given to show Tomasulo's Algorithm is not optimal in reality. An improved algorithm is introduced in the next section. The effectiveness and cost of the new algorithm are addressed. In section 8 we extend the discussion to the cases when Load/Store instructions are taken into consideration. And finally, Section 9 draws together the study and considers its implications for architectural directions.

2. Assumptions

To prove or disprove the optimality of Tomasulo's Algorithm, we compare it with other hardware scheduling algorithms. If none of them can produce wiser execution streams, which consume less clock cycles, Tomasulo's algorithm is the winner. Thus, each of our models is composed of two independent parts. One is a model for a refined system implementing Tomasulo's Algorithm, the object of our optimality proof. The other model, called DATA-DRIVEN, specifies a category containing all dynamic scheduling mechanisms, also including Tomasulo's Algorithm. It captures all the possible behaviors as long as the data dependences are not violated. The input of each model is a program and an initial register file. The output is the time needed to finish the program correctly. The input and output can be modeled as below:

package R/R_PROGRAM (R, N: $\mathbb{N}^+$) *is*
types
  REG_ID=[1..*R*];
  TARGET= *REG_ID*;
  SRC= array[1..2] of *REG_ID*;
  OP_TYPE = {fpadd, fpsub, fpmlt, fpdiv, intFunc};
  INST= [*op*:*OP_TYPE*, *target*:*TARGET*, *src*:*SRC*];
variables
  prog: array[1..*N*] of *INST*;
  finishTime:$\mathbb{N}^+$
definitions

$$\text{lastWriter}(i{:}[0..N], r{:}[1..R]){:}\ [0..N] = \begin{bmatrix} \text{if } i{=}0 \text{ then } 0 \\ \text{else if } prog[i].target{=}r \text{ then } i \\ \text{else lastWriter}(i\text{-}1, r) \end{bmatrix}$$

end package

This model declares the main types and variables, which will be used throughout all models discussed in this paper. It takes R as a parameter which is a positive integer specifying the number of registers. The program can be abstracted as prog, a stream of instructions of length N. Each instruction is made up of an opcode op, a target target, and 2 sources src. Operations range over floating-point addition, subtraction, multiplication, division and fixed-point function. Targets and

sources are described by register indices. The time needed to finish this program is given by a positive integer finishTime. The function lw(i,r) helps to find out the last instruction which writes register r preceding instruction i. It gets the result in a recursive way.

Before looking at the formal proof/disproof, it is helpful to make explicit some of the assumptions underlying all models.

To start with, we are assuming that register renaming is performed in both systems. Renaming eliminates both antidependence, i.e. WAR hazards, and output dependence, i.e. WAW hazards. This basically means all the operands can be got immediately after they are calculated without waiting for its being written back to the register file. Later, we'll consider the case where communication accounts for inneglectable delay.

Another key assumption is that the branch prediction can always give wise results. In other words, both of the systems can take benefits by exploiting ILP across multiple basic blocks without worrying about the penalties brought by branch misprediction. This assumption helps us divide the responsibility between dynamic scheduling and branch prediction.

We also separate instruction and data cache. We assume our caches have so good performance that instruction misses happen very rarely and can be fully hidden. As a result, all instructions are fully pipelined as long as the program hasn't been finished. However, there may still be data cache misses and they form a great impact on the performance of Load/Store.

In addition, we suppose we have separated functional units for FP instructions and integer instructions. However, as there are no critical differences between them, we just regard integer functional units as units with less execution cycles. They share the same register files and compete for the same resources with FP functional units. In our models, we have three types of functional units: floating-point adder, floating-point multiplier and fixed-point unit.

Finally, we assume we have an acceptable compiler. It may not be smart enough to do favor of the hardware by avoiding most of the data dependences. But at least it can make reasonable assignments of registers so that we don't need to worry about running out of registers.

3. The Initial Model

As an initial model, we will consider a case that is as simple as can be. We have infinite hardware. Both functional units and data bus are always ready. Infinite instruction window size and infinite reservation stations are available. Also, we have fast hardware. Any overhead and communication delay can be ignored. Instructions can be issued and executed immediately after it appears in the instruction cache. And operands can be got as soon as they are worked out. This thoroughly unrealistic protocol, to which we give the nickname "utopia", is shown below.

3.1 Model for a generic data-driven system

ModelInil_1 The definition of DATA-DRIVEN system:

System DATA-DRIVEN (R, N: $\mathbb{N}^+$) *is*
import R/R_PROGRAM(R, N)
variables
Completed: array[0..N] of boolean, init {*Completed*[0] = true, *Completed*[i] = false, $1 \le i \le N$};
time : array[0..N] of $\{0\} \cup \mathbb{N}^+$, init 0;
definitions
ready (i:[1..N], j:[1,2]): boolean = *Completed*[lastWriter (i-1, *prog*[i].*src*[j])];
getTime (i:[1..N], j:[1,2]): $\{0\} \cup \mathbb{N}^+$ = *time*[lastWriter (i-1, *prog*[i].*src*[j])];
x = choose $i \in [1..N]$ s.t. $\neg$*Completed*[i] $\wedge$ ready[i,1] $\wedge$ ready[i,2];
waitTime = choose $t \in \{0\} \cup \mathbb{N}^+$;
behavior
if $\neg$*Completed*[x] $\wedge$ ready(x ,1) $\wedge$ ready(x ,2) then
time[x] := max[getTime (x ,1), getTime (x ,2)] + waitTime + duration (*prog*[x].*op*);
Completed[x] := true;
end if
if $\bigwedge_{i=1}^{N}$ *Completed*[i] then
finishTime := $\max_{i=1}^{N}$ *time*[i];
endif
end system

In this model, to keep track of the finish time of each instruction, we use two state arrays: Completed and time, where the boolean array Completed identifies the completed instructions, and the non-negative integer array time represents their finishing time denoted by clock cycles. We don't model the values in the register file, as the things in which we are interested here are the dependence relationship and the execution time instead of the result of the program.

Besides declaration of variables, the model contains the definitions of two auxiliary functions which are used to simplify the following statements. For instruction *i*, the boolean function ready(*i,j*) is defined to be true if and only if its operand *j* has been computed. This is the case when the instruction supposed to compute the operand, given by lastWrite(*i*-1, *prog*[*i*].*src*[*j*]), has already been completed. Similarly, function getTime(*i,j*) fetches the time when operand *j* is ready. This should be the time when the instruction lastWrite(*i*-1, *prog*[*i*].*src*[*j*]), which is responsible for it, is finished. Note that in the special case that lastWrite(*i*-1, *prog*[*i*].*src*[*j*]) = 0, which means that no previous instruction has assigned a value to the register, ready(*i,j*) = *completed*[0] = true and getTime(*i,j*) = *time*[0] =0.

We also use a non-deterministic selection operator **choose** in this model. This operator assigns to x any index *I* in the range [1..*N*] which satisfies the requirement $\neg Completed[i] \wedge \text{ready}[i,1] \wedge \text{ready}[i,2]$. It basically means to non-deterministically choose an instruction whose both operands are already ready. If no such instruction exists, x is assigned an arbitrary value in the specified range. That's why even after the selection, it's necessary to test whether x satisfies the specified requirement. **Choose** is also used to assign a non-negative integer to *waitTime*. That's because even if there are a bunch of ready instructions, functional units can choose to stay idle and just let the instructions wait. This sounds like a waste of resources, but as we'll show later, sometimes it can lead to wiser arrangements and thus better performance. As a result, each instruction can be executed immediately after it is ready, where *waitTime*=0, or wait for some time before it is executed, where *waitTime* is a positive integer. It is these two operations that

introduce the high non-determinism of this system and help us simulate all possible behaviors of all possible algorithms.

As there is not any limitation from hardware, all of the limitation comes from dependence relationship decided by the program. We can use a topological diagram to describe the dependence relationship and the time needed to execute an instruction. To simulate this, the system selects an instruction x which is already ready but has not been completed, completes it and sets its completed time *time*[x] as the sum of the time when its sources are both ready, its waiting time, and the calculation time which is given by the generic function duration. When all of the instructions are completed, the largest finish time is returned as the time needed for the whole program. **Figure** 1.

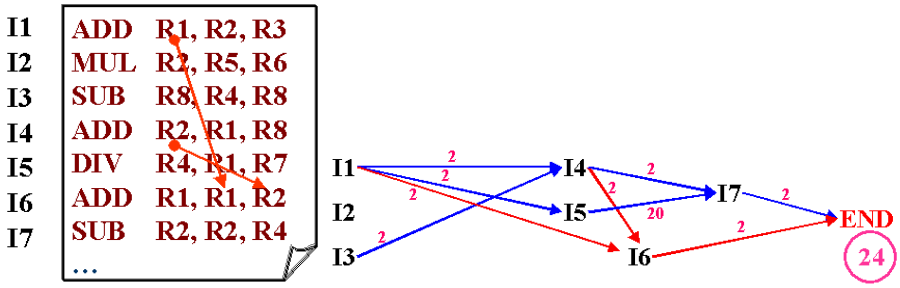


Figure 1

3.2 Model for Tomasulo's Algorithm

ModelInit_2 presents the formal definition of system TOMASULO, which represents a possible implementation of the Tomasulo's Algorithm. Unlike the DATA-DRIVEN system, this system is divided into several subsystems based on distinct parts of the hardware.

We use type *PRODUCER*, which is a tuple of function index and RS index, to denote tags. We assume the same type of functional units share a single reservation station pool. As analyzed in [], RS pooling can give better performance than organizing a RS group for each functional unit. So we just examine the former case.

ModelInit_2 The definition of TOMASULO system

System TOMASULO $(R,N: \mathbb{N}^+)$ *is*
import R/R_PROGRAM(R, N)
type
$PRODUCER = [f:[1..3],\ s:[1..\infty]]$
$R\_TYPE = [busy$:boolean, tag:$PRODUCER$, $time$:$\{0\} \cup \mathbb{N}^+]$;
$SRC\_TYPE =$ array$[1..2]$ of R_TYPE;
$RS\_TYPE =$
$[busy$: boolean, $ready$: boolean, $time : \{0\} \cup \mathbb{N}^+$, $prog : [0..N]$, op:OP_TYPE, $target$:$TARGET$, src:$SRC\_TYPE]$;
$WIN\_TYPE = [busy$: boolean, $prog : [0..N]$, $instru : INST]$;
variables
$instruWin$: array$[1..\infty]$ of WIN_TYPE, init <false, 0,<0,0,0>>;
RS: array$[1..3,\ 1..\infty]$ of RS_TYPE,
init < false,false,0,0,0,0,< false, < 0,0 > ,0 >,< false, < 0,0 > ,0 >>;
$regFile$: array$[1..R]$ of R_TYPE, init < false, < 0,0 > ,0 >;
CDB: array$[1..\infty]$ of R_TYPE, init < false, < 0,0 > ,0 >;
$Completed$: array$[0..N]$ of boolean, init $\{Completed[0] = \text{true},\ Completed[\text{i}] = \text{false},\ 1 \le i \le N\}$;
$time$: array$[0..N]$ of $\{0\} \cup \mathbb{N}^+$, init 0;
$top : [1..N+1]$, init 1;
behavior
$\text{FETCH} \,||\, \left(\underset{FU=1}{\overset{3}{||}} \text{FUNC}(FU) \right) ||\, \left(\underset{r=1}{\overset{R}{||}} \text{WRITERESULT}(r) \right)$;
if $\overset{N}{\underset{i=1}{\wedge}} Completed[i]$ then
$finishTime := \max_{i=1}^{N} time[i]$;
endif
end system

The system has 4 parts of hardware. The instruction window, denoted by instruWin, is composed by several window items. Each window item includes the fields of busy, a boolean describing whether it is occupied by some instruction in the program, prog, the index of the instruction in the program, and instru, the decoding instruction recording the operation code, the target register and the two source registers. CDB and register file both contain infinite items of the same type R_TYPE. This type gives us information on whether it is busy, described by busy, when the data are got, described by time, and which RS item is in charge of its value, described by tag. The difference between CDB and register file is that register file can control its own conduct while CDB is controlled by RS. That's why we have a separate subsystem for register file but not for CDB. For reservation station items, we record more things. Fields included are whether it's occupied, busy, whether the result has already been calculated and ready to transmit, ready, when the calculation is finished, time, which instruction is stored in it, prog, and the opcode, target and two sources in the instruction. Before the program is executed, all items of windows, CDB, registers and RSs are empty.

Like the system DATA-DRIVEN, we define the arrays of Completed and time to keep track of whether and when an instruction is completed. Also, we use the pointer top to keep the instruction index that is going to be fetched from the instruction cache.

In the system, distinct hardware works parallelly. Instruction window gets instructions from the cache whenever there are some empty slots. RS gets instructions from the window, gets operands from CDB or registers files, sends instructions to functional units immediately after they are ready, and broadcasts the results on CDB as soon as they are available. At the same time, registers get the

matched value from CDB whenever it's ready. This process repeats itself until at some point all of the instructions are finished. Then the longest time to finish an instruction is returned as the program's finish time.

ModelInit_2_1 The definition of FETCH system

```
System FETCH is
 definitions
   winTail = getAvailableWin;
 behavior
   if top ≤ N ∧ winTail ≠ −1 then
    instruWin[winTail].busy = true;
    instruWin[winTail].prog = top;
    instruWin[winTail].instru = prog[top];
    top := top + 1;
   end if
end system
```

ModelInit_2_1 described the first step to handle with an instruction, to fetch it from the instruction cache and put it into the window. Using the function getAvailableWin can help us maintain the instruction queue in the window and get an available window item. As the window size is infinite in the initial model, getAvailableWin can return a usable window with no exception.

ModelInit_2_2 The definition of FUNC system:

```
System FUNC(FU:[1..3]) is
behavior
  (||_{S=1}^{∞} ISSUE[FU,S]) || (||_{S=1}^{∞} SNOOPER[FU,S]) || EXEC[FU] || (||_{S=1}^{∞} BROADCAST[FU,S]);
end system
```

Reservation Station is the central part of Tomasulo's Algorithm. RS by itself has four subsystems: ISSUE, SNOOPER, EXEC and BROADCAST.

ModelInit_2_2_1 The definition of ISSUE system:

```
System ISSUE[FU : [1..3], S : ℕ+] is
 definitions
   src(i : ℕ+, j:[1..2]) : R_TYPE
     = if ¬regFile[instruWin[i].instru.src[j]].busy
       then <false,< 0,0 >, regFile[instruWin[i].instru.src[j]].time>
       else regFile[instruWin[i].instru.src[j]];
   winHead:{0} ∪ ℕ+ = min_{win=1}^{∞} {instruWin[win].busy};
   issueReady:boolean = winHead ≠ 0 ∧ instruWin[winHead].busy ∧ instruWin[winHead].instru.op ∈ operation(FU)
 behavior
   if ¬RS[FU,S].busy ∧ issueReady then
       RS[FU,S] :=< true, false, 0, instruWin[winHead].prog,
                    instruWin[winHead].instru.op, instruWin[winHead].instru.target,
                    src(winHead,1), src(winHead,2)>;
       regFile[instruWin[nextWin].target] :=< true,< FU,S >,0 >;
   end if
end system
```

In the ISSUE subsystem described by **ModelInit_2_2_1**, when there is an empty slot in the RS pool for a certain functional unit and the instruction window is not empty, the type of the oldest unissued instruction in the window will be compared with the functional unit type. If they match well, the instruction will be fetched and records for both RS and its target register are updated. If its operand is already ready when issuing, the *time* field is filled according to the same field of the register. If not, the *busy* field is set to true, and the *tag* field is copied from the register file.

ModelInit_2_2_2 The definition of SNOOPER system:

$$
\begin{array}{l}
\textit{System}\ \ \mathrm{SNOOPER}(FU:[1..3], S:\mathbb{N}^+)\ \textit{is} \\
\quad \textit{behavior} \\
\quad\quad \|_{j=1}^{2}\ \|_{bus=1}^{\infty}\ \text{if } RS[FU,S].busy \wedge RS[FU,S].source[j].busy \wedge CDB[bus].busy \\
\quad\quad\quad\quad \wedge RS[FU,S].source[j].tag = CDB[bus].tag \ \text{then} \\
\quad\quad\quad\quad\quad RS[FU,S].source[j].time := CDB[bus].time; \\
\quad\quad\quad\quad\quad RS[FU,S].source[j].busy := \text{false}; \\
\quad\quad\quad\quad \text{endif} \\
\textit{end system}
\end{array}
$$

While issuing instructions, RS also keeps an eye on whether there are some operands ready in CDB by comparing the tags. If so, it will get the value and the available time from CDB and set it to be not busy.

When an instruction has been finished and the operands are ready, RS will send it to an empty functional unit. After that, the *ready* field is set to true, indicating the result is ready to put on the CDB. As functional units are infinite, an instruction can begin to be calculated as soon as both of its operands are ready. As a result, the time when the result is available is the time when both of its operands are ready plus the time needed for the particular operation.

ModelInit_2_2_3 The definition of EXEC system

$$
\begin{array}{l}
\textit{System}\ \ \mathrm{EXEC}(FU:[1..3])\ \textit{is} \\
\quad \textit{definitions} \\
\quad\quad \mathrm{enabled}(f:[1..3], s:\mathbb{N}^+):\mathrm{boolean} \\
\quad\quad\quad = RS[f,s].busy \wedge \bigwedge_{j=1}^{2} \neg RS[f,s].source[j].busy; \\
\quad\quad \mathrm{e} = \text{choose}\ \ k:\mathbb{N}^+\ \ \text{s.t.}\ \ \mathrm{enabled}(FU,k); \\
\quad\quad \mathrm{maxtime}(f:[1..3], s:\mathbb{N}^+):\mathrm{int} = \max_{j=1}^{2} RS[f,s].src[j].time; \\
\quad \textit{behavior} \\
\quad\quad \text{if enabled}(FU,\mathrm{e})\ \text{then} \\
\quad\quad\quad RS[FU,\mathrm{e}].time := \mathrm{maxtime}(FU,\mathrm{e}) + \mathrm{duration}(RS[FU,\mathrm{e}].op); \\
\quad\quad\quad RS[FU,\mathrm{e}].ready := \text{true}; \\
\quad\quad \text{endif} \\
\textit{end system}
\end{array}
$$

Finally, RS is in charge of sending the calculated result from the functional unit to an inactive bus, which will broadcast it to all items in register file and RS. In this model, function getAvailableBus is used to find an inactive bus. Again, as we have unlimited bandwidth, and transmission delay can be ignored, the result can be instantly broadcast and got by other hardware. Therefore, no extra time is needed for this step and the finish time of the instruction is just the time when the calculation is completed. After broadcast, the resources used for this instruction in window buffer and RS are

released. As there is infinite bandwidth and functional units, our model doesn't reset them to be idle and this will not affect the system.

ModelInit_2_2_4 The definition of BROADCAST system:

```
System BROADCAST(FU : [1..3], S : ℕ+) is
 definitions
   bus : ℕ+ = getAvailableBus;
 behavior
   if RS[FU,S].ready ∧ ¬CDB[bus].busy then
        CDB[bus].busy := true;
        CDB[bus].tag: =< FU,S >;
        CDB[bus].time: = RS[FU,S].time;
        time[RS[FU,e].prog] := RS[FU,S].time;
        Completed[RS[FU,e].prog] := true;
        RS[FU,S].ready : = false;
        RS[FU,S].busy : = false;
        ||_{win=1}^{∞} if instruWin[win].prog = RS[FU,S].prog the
              instruWin[win].busy = false;
            endif
   endif
end system
```

In the subsystem WRITERESUTL, which presents the conduction of register file, each register compares the tag of an active bus with its own tag and updates its value when they match.

ModelInit_2_3 The definition of WRITERESULT system

```
System WRITERESULT(r : ℕ+) is
 behavior
    ||_{bus=1}^{∞} if CDB[bus].busy ∧ regFile[r].busy ∧ regFile[r].tag = CDB[bi
        regFile[r].time : = CDB[bus].time;
        regFile[r].busy : = false;
       end if
end system
```

3.3 Proof

To prove Tomasulo's Algorithm is optimal, we need to show that all of the other instruction execution order cannot give better performance than that produced by Tomasulo's Algorithm, which means, finishTime in system DATA-DRIVEN is no less than finishTime in system TOMASULO. Actually, we can prove a stronger conclusion: the finish time of each instruction in the former system is no less than that in the latter system.

In system DATA-DRIVEN, the finish time of an instruction is given by the equation:

$$time_D[\mathrm{x}] = \max[\,\mathrm{getTime}(\mathrm{x},1), \mathrm{getTime}(\mathrm{x},2)] + \mathrm{waitTime} + \mathrm{duration}(prog[\mathrm{x}].op)$$
$$= \max_{j=1}^{2} time_D[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[\mathrm{j}])] + \mathrm{waitTime} + \mathrm{duration}(prog[\mathrm{x}].op)$$

As functional units are infinite, any instruction can be executed as soon as its operands are ready. It is not necessary to wait. So we have:

$$time_D[\mathrm{x}] = \max_{j=1}^{2} time_D[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[\mathrm{j}])] + \mathrm{waitTime} + \mathrm{duration}(prog[\mathrm{x}].op)$$
$$\geq \max_{j=1}^{2} time_D[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[\mathrm{j}])] + \mathrm{duration}(prog[\mathrm{x}].op)$$

On the other hand, in the system TOMASULO, we can get the finish time of an instruction from the description in the model. Before doing it, we first prove an obvious but important fact:

$$\text{Suppose } \mathrm{x} = RS[FU, \mathrm{e}].prog$$
$$RS[FU, \mathrm{e}].op = instruWin[\mathrm{winHead}].instru.op$$
$$(\text{where } instruWin[\mathrm{winHead}].prog = RS[FU, \mathrm{e}].prog = \mathrm{x})$$
$$= prog[\mathrm{x}].op$$

For the same reason:

$$RS[FU, \mathrm{e}].target = prog[\mathrm{x}].target$$

Thus the finish time of an instruction depends on the time its two operands are ready and its own execution time.

$$time_T[\mathrm{x}] = RS[FU, \mathrm{e}].time$$
$$= \mathrm{maxtime}(FU, \mathrm{e}) + \mathrm{duration}(RS[FU, \mathrm{e}].op)$$
$$= \max_{j=1}^{2} RS[FU, \mathrm{e}].src[j].time + \mathrm{duration}(prog[\mathrm{x}].op)$$

By tracing the flow of data and instruction, we can find that there are two sources for the available time of an instruction's operand. One comes from the register. It happens if the operand has already been ready in the register file when the instruction is issued. Thus we have:

$$RS[FU, \mathrm{e}].src[j].time$$
$$= regFile[instruWin[i].instru.src[j]].time$$
$$(\text{where } instruWin[i].prog = RS[FU, \mathrm{e}].prog = \mathrm{x})$$
$$= regFile[prog[\mathrm{x}].src[j]].time$$
$$= \begin{cases} 0 & \text{no previous instructions write back the instruction} \\ CDB[bus].time & \text{once exsits } bus,\ CDB[bus].tag = regFile[prog[\mathrm{x}].src[j]].tag) \end{cases}$$
$$= RS[FU', S].time \quad (\text{where } < FU', S >= CDB[bus].tag = regFile[prog[\mathrm{x}].src[j]].tag)$$
$$= time_T[RS[FU', S].prog]$$

For the other possibility, that the source is not available at issue point, RS gets the time directly from the CDB after the source value has been figured out.

$$
\begin{aligned}
& RS[FU, \mathrm{e}].src[j].time \\
&= CDB[bus].time \\
&\quad (\text{where } CDB[bus].tag = RS[FU, \mathrm{e}].src[j].tag = regFile[prog[\mathrm{x}].src[j]].tag) \\
&= RS[FU', S].time \quad (\text{where } < FU', S >= CDB[bus].tag = regFile[prog[\mathrm{x}].src[j]].tag) \\
&= time_T[RS[FU', S].prog]
\end{aligned}
$$

Thus, as long as there are some former instructions that write back the source register, the available time for the source equals the time the source is broadcast, and in turn equals the time the source is calculated. Also, we can prove:

$$
\begin{aligned}
&\because < FU', S >= regFile[prog[\mathrm{x}].src[j]].tag \\
&\therefore prog[\mathrm{x}].src[j] = RS[FU', S].target = prog[RS[FU', S].prog].target \\
&\therefore RS[FU', S].prog = \mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[j]) \\
&\therefore time_{\mathrm{T}}[\mathrm{x}] = \max_{j=1}^{2} time_{\mathrm{T}}[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[j])] + \mathrm{duration}(prog[\mathrm{x}].op)
\end{aligned}
$$

By induction, we assume that for all the instructions with the index less than *n*, it can be finished in TOMASULO system no later than in DATA-DRIVEN system. Thus for instruction *n*, we have

$$
\begin{aligned}
time_{\mathrm{D}}[n] &\geq \max_{j=1}^{2} time_{\mathrm{D}}[\mathrm{lastWriter}(n-1, prog[n].src[j])] + \mathrm{duration}(prog[n].op) \\
&\geq \max_{j=1}^{2} time_{\mathrm{T}}[\mathrm{lastWriter}(n-1, prog[n].src[j])] + \mathrm{duration}(prog[n].op) \\
&= time_{\mathrm{T}}[n]
\end{aligned}
$$

Also, due to initialization, $time_{\mathrm{D}}[0]=time_{\mathrm{T}}[0]=0$, so we have that for all instructions, TOMASULO system can finish them no later than DATA-DRIVEN system. In conclusion, Tomasulo's Algorithm is optimal.

4. The Bus Model

Before tackling the general case, let us first drop an unrealistic but not critical restriction used in the utopia model, which says that data can be transmitted throughout the algorithm with no delay. This assumption is not only made by our initial model, but also by Tomasulo's Algorithm itself. However, it is currently not valid, and as the computer architectures are getting more and more complex, less and less can we expect it to be so.

The long round-trip time can lead to longer completion time. To make things easier and cleaner, we just examine the on-chip wire delay after an operation has been performed. Other kinds of transmission can be coped with in exactly the same way. In this more realistic model, we still suppose we have infinite bandwidth. However, a certain amount of extra cycles are needed to transmit data around the chip, and the transmission time is unique. To make it fare, this is applicable to both systems. The adjustment is only needed to be made to **Model_1** and **Model_2_2_4**

After the modification, transmission delay is added to the finish time of an instruction in both systems. Also, in system TOMASULO, the available time for the data broadcast on CDB is increased by the transmission time. Other conductions of the model are just the same.

Model 1_bus0 The definition of DATA-DRIVEN system:

```
System DATA-DRIVEN (R, N: ℕ+) is
  …
behavior
 if ¬Completed[x] ∧ ready[x,1] ∧ ready[x,2] then
      time[x] := max[getTime(x,1), getTime(x,2)] + waitTime + duration(prog[x].op) + transmitTime;
      Completed[x] := true;
 end if
 …
end system
```

Model 2.2.4_bus0 The definition of BROADCAST system:

```
System BROADCAST(FU : [1..3], S : ℕ+) is
 ...
 behavior
   ||_{S=1}^{∞}  if RS[FU,S].ready ∧ ¬CDB[bus].busy then
          ...
          CDB[bus].time := RS[FU,S].time + transmitTime;
          time[RS[FU,e].prog] := RS[FU,S].time + transmitTime;
          ...
        endif
end system
```

In the new model, the optimality of Tomasulo still stands. We can prove in the same way that

$$\begin{aligned} time_{\mathrm{D}}[\mathrm{x}] &\geq \max_{j=1}^{2} time_{\mathrm{D}}[\mathrm{lastWrite}(\mathrm{x}-1, prog[\mathrm{x}].src[j])] + \mathrm{duration}(prog[\mathrm{x}].op) + \mathrm{transmitTime} \\ &\geq \max_{j=1}^{2} time_{\mathrm{T}}[\mathrm{lastWrite}(\mathrm{x}-1, prog[\mathrm{x}].src[j])] + \mathrm{duration}(prog[\mathrm{x}].op) + \mathrm{transmitTime} \\ &= time_{\mathrm{T}}[\mathrm{x}] \end{aligned}$$

However, our assumption that the two systems have the same transmission delay may not be true in two cases.

- One case happens when forwarding is used inside functional units. In such a way, a result coming from a functional unit can be used directly by the following depending instruction without waiting for the bus transmission. Furthermore, some optimization can be done by redundant execution of an instruction.

  A simple example is shown in **Figure 2**. Suppose we have 2 adders. Addition can be done in 1 cycle and every bus transmission counts for 1 cycle. With data forwarding, several cycles can be saved on transmission and the system can finish the program block in 5 cycles. However, if the system is smarter and duplicates the first instruction in both of the adders, another cycle is saved in transmission.

  No matter which algorithm the system uses, Tomasulo's algorithm is left behind as it has to pay for the transmission delay.

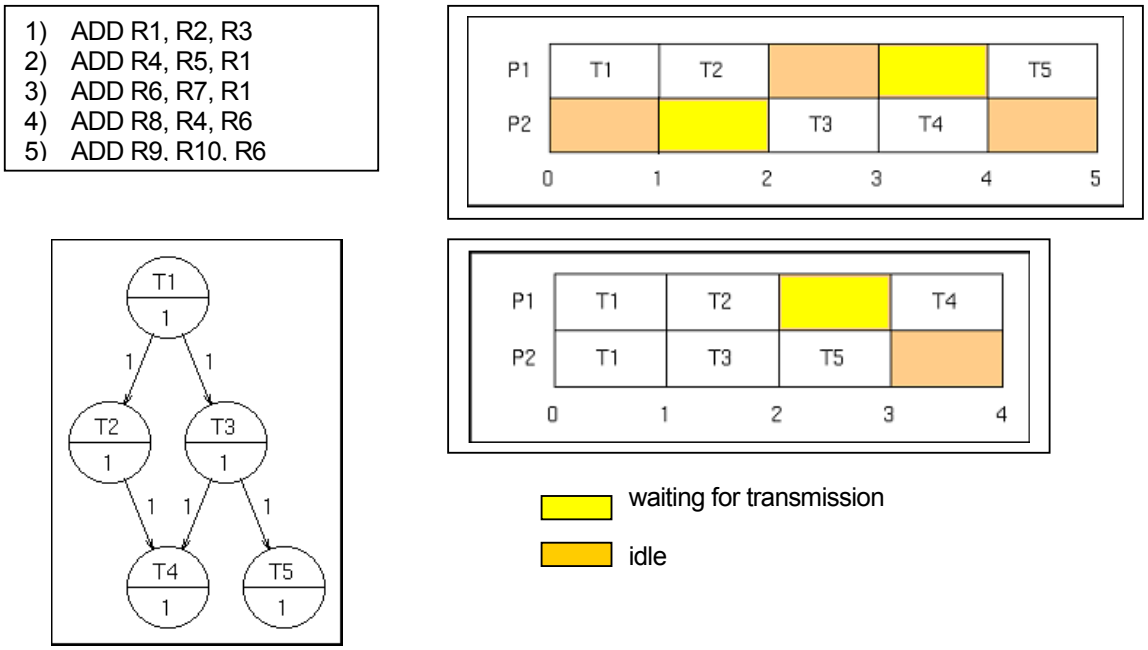


Figure 2

- The other case is data-flow system. In such system there may or may not be transmission delay between two instructions in the same data stream. And the transmission delay is not fixed. Due to its complexity and unpopularity, discussion of this problem is beyond our range.

5. Finite Instruction Window

In the previous models, we have infinite hardware resources thus any instructions can be fetched, issued, executed and transmitted without any necessary wait. This assumption is clearly unrealistic. As Tomasulo's Algorithm uses in-order issue and out-of-order execution, hardware can be divided into two categories. One includes the window buffer, which is basically a queue. The other includes reservation stations, functional units, and CDB, where instructions can be moved in and out in any arbitrary order.

In this section, we discuss the first category. We'll show that the limitation on the window size will not influence the algorithm's optimality. The main reason is that the issue order is determined by the program itself with no way to use dynamic execution information. Both systems have to fetch instructions from the window in the same mechanism, indicating the same fetching order.

This model set evolves from the second model. It characterizes the effect of importing finite instruction window. All of the other assumptions are kept except that instruction window size alters from infinite to a positive number.

ModelWin_1 The definition of DATA-DRIVEN system

```
System DATA-DRIVEN (R, N: ℕ+) is
...
definitions
  ...
  inWindow(i : [1..N]) : boolean = |{j | 1 ≤ j ≤ i ∧ ¬Completed[j]}| ≤ I
  replacedInstru(i : [1..N]) : [1..N] = j
    s.t. Completed[j] ∧ |{k | k ∈ [1..N] ∧ Completed[k] ∧ time[k] ≤ time[j]}| = i − I
  inWinTime(i : [1..N]) : {0} ∪ ℕ+
    = if i ≤ I then 0
      else time[replacedInstru(i)]
behavior
 if ¬Completed[ x ] ∧ inWindow[x] ∧ ready(x ,1) ∧ ready(x ,2) then
      time[x] : = max[inWinTime(x), getTime( x ,1), getTime( x ,2)] + waitTime+ duration (prog[ x ].op);
      Completed[ x ] : = true;
 end if
 ...
end system
```

In the DATA-DRIVEN system, as now not all of the instructions can be issued in the beginning, it's possible that for some of the instructions, although both of its operands are already ready, it's still waiting for window buffer and thus cannot be executed immediately. Consequently, we need to find the maximum of its ready time and its fetching time. It is at this point that the instruction is sent to the execution unit and the calculation begins.

Two auxiliary functions are added to justify whether an instruction has already been in the window and what's the exact time of its being sent to the window. The first function is based on the observation that when the *i*th instruction has been put into the window, all of its previous instructions are either finished or in the window. As unfinished instructions are no more than the window size, the number of the completed previous instructions is at least *i-I*. Also, we know that for each instruction with an index larger than *I*, the earliest time it is sent to the window is exactly the time when *i-I* instructions in total have been finished. Another auxiliary function replacedInstru is used to find the last finished instruction among those *i-I* instructions. Its finish time is the time when the *i*th instruction gets the window item.

In the Tomasulo's Algorithm, we add a new item to *WIN_TYPE* to keep records of when the specific window buffer is idle and can be assigned to a new instruction.

ModelWin_2 The definition of TOMASULO system

```
System TOMASULO (R,N: ℕ+) is
 import R/R_PROGRAM(R, N)
 type
  WIN _ TYPE = [busy : boolean, time : {0} ∪ ℕ+, prog : [0..N], instru : INST];
 variables
  instruWin : array[1..∞] of WIN _ TYPE, init <false, 0,0,<0,0,0>>;
 ...
end system
```

When issuing, this time is copied to the *time* field in RS for comparison later with the instruction's ready time.

ModelWin_2_2_1 The definition of ISSUE system:

```
System ISSUE[FU : [1..3], S : ℕ+] is
 ...
 behavior
   if ¬RS[FU, S].busy ∧ issueReady then
       RS[FU, S] :=< true, false, instruWin[winHead].time, instruWin[winHead].prog,
                    instruWin[winHead].instru.op, instruWin[winHead].instru.target,
                    src(winHead, 1), src(winHead, 2)>;
       regFile[instruWin[nextWin].target] :=< true, < FU, S >, 0 >;
   end if
end system
```

ModelWin_2_2_3 The definition of EXEC system

```
System EXEC(FU : [1..3]) is
 definitions
   ...
   maxtime(f : [1..3], s : ℕ+) : ℕ+ = max[ max_{j=1}^{2} RS[f, s].src[j].time, RS[f, s].time ];
 ...
end system
```

Finally, when an instruction is finished and its window item is released, its finish time is noted down and that's the fetch time for the next instruction

ModelWin_2_2_4 The definition of BROADCAST system:

```
System BROADCAST(FU : [1..3], S : ℕ+) is
 ...
      ||_{win=1}^{∞} if instruWin[win].prog = RS[FU, S].prog then
                       instruWin[win].busy = false;
                       instruWin[win].time = RS[FU, S].time + transmitTime;
                     endif
      endif
end system
```

Using the results in 3.1, we can easily get the finish time of each instruction in DATA-DRIVEN system:

$$time_{\mathrm{D}}[\mathrm{x}] \geq \max\left[time_{\mathrm{D}}[\mathrm{lastWriter}(\mathrm{x}\text{-}1, prog[\mathrm{x}].src[1])], time_{\mathrm{D}}[\mathrm{lastWriter}(\mathrm{x}\text{-}1, prog[\mathrm{x}].src[2])], \mathrm{inWinTime}(\mathrm{x})\right] + \mathrm{duration}(prog[\mathrm{x}].op) + \mathrm{transmitTime}$$

$$\mathrm{inWinTime}(\mathrm{x}) = \begin{cases} 0 & \mathrm{x} \leq I \\ time_{\mathrm{D}}[j] \ (\text{where } \left|\{k \mid k \in [1..N] \wedge time_{\mathrm{D}}[k] \leq time_{\mathrm{D}}[j]\}\right| = i - I) & x > I \end{cases}$$

By tracing we get the finish time in TOMASULO system as:

$$time_{\mathrm{T}}[\mathrm{x}] = \max\left[time_{\mathrm{T}}[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[1])], time_{\mathrm{T}}[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[2])], RS[FU, \mathrm{e}].time\right] + \mathrm{duration}(prog[\mathrm{x}].op) + \mathrm{transmitTime}$$

$$= \max\left[time_{\mathrm{T}}[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[1])], time_{\mathrm{T}}[\mathrm{lastWriter}(\mathrm{x}-1, prog[\mathrm{x}].src[2])], instruWin[w].time\right] + \mathrm{duration}(prog[\mathrm{x}].op) + \mathrm{transmitTime} \quad (\text{where } instruWin[w].prog = RS[f, s].prog = \mathrm{x})$$

$$instruWin[i].time = \begin{cases} 0 & \mathrm{x} \le I \\ RS[f, s].time & x > I \end{cases}$$

$$= time_{\mathrm{T}}[j] \quad (\text{where } instruWin[w].prog' = RS[f, s].prog = j)$$

Again, according to the window handling mechanism and the behavior of function getAvailableWin in the TOMASULO system, we have

$$\left|\{k \mid k \in [1..N] \wedge time_{\mathrm{T}}[k] \le time_{\mathrm{T}}[j]\}\right| = i - I$$

It is easy to understand that the indices of all the instructions that are finished before fetching instruction *n* is less than *n.* Thus, it holds that $time_{\mathrm{T}}[k] \le time_{\mathrm{D}}[k]$, that is *instruWin*[*i*].*time* in TOMASULO system is less than or equal to inWinTime(x) in DATA-DRIVEN system. So once more TOMASULO system can finish each instruction in shortest time.

6. The Limited Resource Model

While limitation to the first category hardware cannot shake the optimality, that to the second category can bring disasters to the algorithm. They can have important implications for the execution order of instructions and in turn for the optimality of the algorithm.

In this model, neither functional units nor bus bandwidth is infinite. When these restrictions are brought in, a lot of changes are needed to be made. The most significant difference is, as a result, sometimes it is inevitable for an instruction to wait for resources.

6.1 Counterexample for Tomasulo's Algorithm

In face of resources shortage, TOMASULO system uses a greedy algorithm. That is, the ready instructions are issued, executed and broadcast whenever they find some available resources. On the other hand, DATA-DRIVEN system may be patient enough to wait for some time, which can lead to a wiser choice. Tomasulo's Algorithm is not optimal any more.

We can find similar counterexamples to disprove the optimality of Tomasulo's Algorithm at each level, including issuing, executing and writing back. Here we use the choice made on instruction execution as an example.

The first counterexample shows why the greedy algorithm cannot win. In order to simplify, we don't consider transmission delay and in-window-time in this example. **Figure 3**.

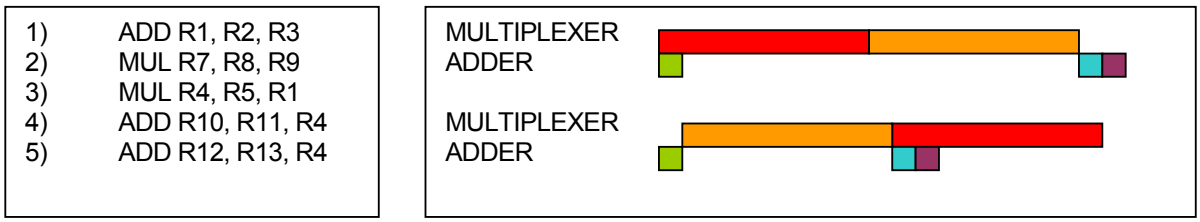


Figure 3

In this example, suppose we have only one adder and one multiplier. Also, we assume addition can be finished in 2 cycles while multiplication needs 20 cycles. At first only instruction 1 and 2 are ready. If we follow Tomasulo's Algorithm, we'll execute them parellelly. Instruction 3 are ready at the 3rd clock cycle but have to wait until the 20th cycle when instruction 2 is finished. As instruction 4 and 5 are both dependent on instruction 3, they can't begin until the 42nd cycle. The whole process finishes in the 44th cycle.

However, if the system is patient and smart enough, it may delay the execution of instruction 2. Then at the 3rd clock cycle, both the 2nd and the 3rd instructions are ready and it once again wisely chooses instruction 3 to execute. As a result, instruction 4 and 5 can be executed parallely with instruction 2 after instruction 3 is finished. The whole process lasts for 42 cycles and 2 clock cycles are saved.

The second case is about an implementation detail. Suppose two instructions are ready at the same time. Decision is needed to be made on which instruction is to be executed next. Tomasulo's Algorithm doesn't explicitly state how to make this kind of choices. This example demonstrates that different choices can lead to different finish time. **Figure 4**.

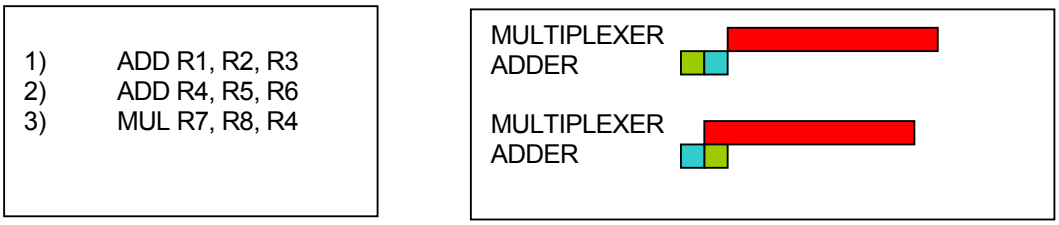


Figure 4

In this example, the first two instructions are ready at the same time. However, there is another instruction dependent on the 2nd instruction. If we simply give priority to older instructions, then the 1st instruction will be chosen and both instruction 2 and instruction 3 have to wait. On the contrary, if the resources are assigned to the more urgent instruction, instruction 2 in this example, the block can be finished in 22 cycles instead of 24 cycles.

Actually, when we have infinite resources, scheduling is rather a mapping problem than a scheduling problem. The key point here is to map computation tasks into infinite computation resources. Tomasulo always maps "maximal" tasks, as there is no wait and thus no waste. When resources are limited, some trade-off must be made. Now the problem becomes how to use the resources in a delicate and smart way. Temporary waiting, which seems to be a waste at the first glance, may bring wiser decision and better performance. Lacking the mechanism of waiting and comparing is the reason why Tomasulo's algorithm loss.

6.2 Model for Tomasulo's Algorithm under restricted resources

Although in this section we don't need to use formal models for proof, we'd like to give out the model for Tomasulo's algorithm, as they will act as the basis in describing our improvement method in the next section.

Resource limitation brings a surgery to our model from parallel system to sequential simulation system. This is done by using a variable *t* to indicate the current clock cycle. The reason is that it is easier to catch the resource waiting by simulating using a time description variable.

As functional units are limited, we add the array *funcUnit* to describe whether the resources are busy, whether the calculation has been finished, when they begin to be occupied and which RS item occupies them. In the *RS*_TYPE, we add a new field to keep track of whether it has been sent to functional units. We also add another sub-system to describe the conduction of the CDB when a transmission has ended.

ModelRes_2 The definition of TOMASULO system

System TOMASULO $(R, N, K, U: \mathbb{N}^+, FUNL: \text{array } [1..2] \text{ of } \mathbb{N}^+)$ *is*
import R/R_PROGRAM(R, N)
type
...
$RS\_TYPE = [exeu: boolean; ...]$
$FUN\_TYPE = [busy: \text{boolean}, ready: \text{boolean}; tag: PRODUCER, timeStamp: \{0\} \cup \mathbb{N}^+];$
variables
…
$funcUnit$: array[1..3, 1..$\max_{j=1}^{3} FUNL[j]$] of FUN_TYPE init <false,false,<0,0>,0>;
t: $\{0\} \cup \mathbb{N}^+$ init 0;
behavior
$t := 0;$
while $\bigvee_{i=1}^{N} \neg Completed[i]$ do
$\text{FETCH}(t) \,||\, \left(\|_{FU=1}^{3} \text{FUNC}(t, FU) \right) \|| \left(\|_{bus=1}^{K} \text{DEACTIVATE}(t, bus) \right) \|| \left(\|_{r=1}^{R} \text{WRITERESULT}(r) \right);$
$t := t + 1;$
end while
return t;
end system

ModelRes_2_2_3 The definition of EXEC system

System EXEC$(t: \{0\} \cup \mathbb{N}^+, FU: [1..3])$ *is*
definitions
…
enabled$(s: \mathbb{N}^+)$: boolean
$= RS[FU, s].busy \wedge \bigwedge_{j=1}^{2} \neg RS[FU, s].source[j].busy \wedge \neg RS[FU, s].exeu;$
freeFunc = choose $j \in [1..FUNL[FU]]$ *s.t.* $\neg funcUnit[FU, j].busy;$
behavior
if enabled(e) $\wedge \neg funcUnit[FU, \text{freeFunc}].busy$ then
$funcUnit[FU, \text{freeFunc}].busy := \text{true};$
$funcUnit[FU, \text{freeFunc}].tag :=< FU, \text{e} >;$
$funcUnit[FU, \text{freeFunc}].timeStamp := t;$
$RS[FU, \text{e}].exeu := \text{true};$
endif
end system

In the execution subsystem, an idle functional unit is selected when an instruction has got both of its operands. If such an idle unit exists, it is initialized and the execution begins. Or else the instruction has to wait.

When the execution is finished, the *ready* field of the functional unit is set to true to indicate that it can be put on the CDB. After it gets an idle bus, the data is put there; the RS item and the functional unit are released. Again, to consider transmission delay, the available time of the data on CDB is the current time plus transmission time.

ModelRes_2_2_4 The definition of BROADCAST system:

```
System BROADCAST(t : {0} ∪ ℕ+, FU : [1..3], u : [1..FUNC[FU]]) is
 definitions
  freeBus = choose j ∈ [1..K] s.t. ¬CDB[j].busy;
 behavior
  if funcUnit[FU,u].busy ∧ funcUnit[FU,u].timeStamp + duration(FU) = t then
    funcUnit[FU,u].ready := true;
  endif
  if funcUnit[FU,u].ready ∧ ¬CDB[freeBus].busy then
    CDB[bus].busy := true;
    CDB[bus].tag := funcUnit[FU,u].tag;
    CDB[bus].time := t + transmitTime;
    funcUnit[FU,u].busy := false;
    funcUnit[FU,u].ready := false;
    RS[FU,S].busy := false;
  endif
end system
```

Subsystem DEACTIVATE is added to deactivate a bus when the transmission is finished. This is also the time when the instruction is completed. The instruction is set to be completed and the finish time is recorded. Also, the instruction window and CDB are released.

ModelRes_2_4 The definition of DEACTIVATE system:

```
System DEACTIVATE(t : {0} ∪ ℕ+, bus : [1..K]) is
 behavior
  if t = CDB[bus].time ∧ CDB[bus].busy then
      time[RS[CDB[bus].tag].prog] := t;
      Completed[RS[CDB[bus].tag].prog] := true;
      CDB[bus].busy := false;
      ||_{win=1}^{∞} if instruWin[win].prog = RS[CDB[bus].tag].prog then
                      instruWin[win].busy = false;
                    endif
  endif
end system
```

6.3 Optimality under limitation of instruction issue

The final assumptions we want to drop are those related to instruction issue. In all of the previous models, we assume we can issue as many instructions as we can. However, this is definitely not true. The first constraint comes from the restriction of reservation stations. The number of issued instructions can by no means exceed the number of RS items. Another constraint is also caused by hardware capacity. As a lot of work needs to be done at issue point, there is an upper bound of how many instructions the processor can issue even in multiple issue architectures.

Issue capability decides how much we can look ahead when scheduling. While best choice depends on a global view, the system suffers from the local view brought about by issue limitation. Our first example in section 6.1, again, can serve as a good example here. **Figure 5**.

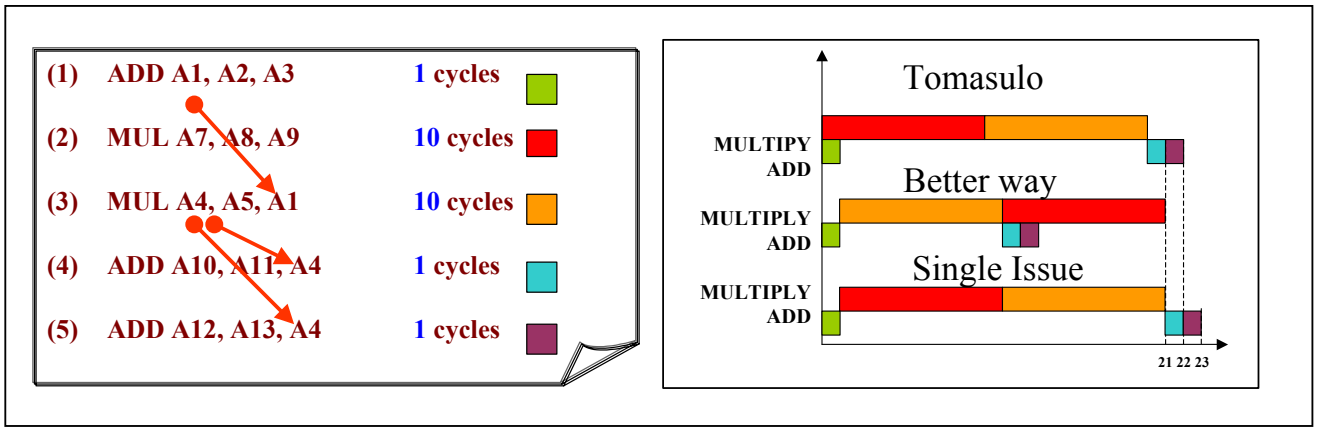


Figure 5

In this example it is supposed that we can only issue one instruction per cycle. Even using our improved algorithm, we'll choose the 2nd instruction to execute first. That's because when this decision is made, in the 2nd cycle, the 3rd instruction has not entered RS so the system doesn't know its existence at all!

Unlike those problems stated above, there seems no way to enhance the algorithm optimality under this situation. Nonetheless, in state-of-art architectures, issue speed outstrips the dispatch speed and the beast is usually fed up quite well. Thus the issue limitation cannot influence the performance too much.

7. Improvement

7.1 The NP-Completeness of the underlying problem

The underlying problem which Tomasulo's algorithm tries to solve is basically a scheduling problem. We have known that scheduling is a difficult problem. Various special cases have been proved to be NP hard or NP complete. Most of the complexity of scheduling can be assigned to the conjunction of two types of constraints.

- Dependence constraints, which express the fact that some computations must be executed in a specified order if the meaning of the original program is to be preserved. This kind of constraints are usually expressed as a dependence graph (DG).

- Resource constraints, which express the fact that the number of simultaneous operation at any given time is limited by the available resources in the target computer

While either of the two constraints can be tackled easily, the combination brings the difficulty. Instruction scheduling for a single-issue processor is NP-complete if there is no fixed bound on the maximum latency [10]. Optimal scheduling of a job system which has two functional units, each of which specializes a disjoint subset of the instruction set, was proved to be NP-complete even if the precedence graph is a forest [11]. These scheduling problems are essentially similar to the job-shop scheduling problem. The generalization of the classical scheduling problem, in which there are more than 2 functional units, has also been shown to belong to the class of NP hard optimization problems. [12][13][14]. A summary of the complexity and optimization of scheduling problems can be found in [23][15][16][17][18][19].

In conclusion, given arbitrary functional units and arbitrary instructions, the problem of finding the optimal execution order and functional units assignment is NP-Complete. Therefore, it is unlikely that there exits a polynomial time algorithm which gives an optimal solution to the general problem. This leads to the belief that we should take a heuristic or approximation algorithm approach, rather than to find an optimal scheduling.

7.2 The improved schedule algorithm

However, the impossibility of finding an optimal algorithm does not imply that Tomasulo's Algorithm is the best way. Tomasulo's Algorithm uses a greedy algorithm to give an approximate answer. One of its obvious defects is that it only uses previous dependence information in its greedy function, leaving the subsequent dependence information ignored. Actually, it is exactly the afterward information that helps decide how urgent the instruction is and whether it should be given higher priority. This indicates there is still a room for improvement. And the breakthrough can lies in how to wisely use the afterward information. Such information includes chain length, dependants number, etc. [8] Here we present an improvement which exploits the information to a great extend. The algorithm is oriented to limited functional units. The same improvement can be made to CDB.

In our improvement, we still use greedy algorithm, as it is the most practical algorithm to hardware. We base our decision on the function defining the least penalty. When a functional unit becomes available, the instruction leading to highest penalty will win it. It's even possible to leave the unit idle and wait for an instruction with larger penalty, instead of executing a ready low-penalty instruction.

Two values for each instruction are used to calculate the least penalty. They are *earliest* and *latest*, denoting the earliest time the instruction may be finished and the latest time the instruction should be finished without prolonging the program's execution. Both of them are adjusted dynamically according to the current execution stream as well as the backward and forward dependence relationship. In turn, the penalty function is also a snapshot of the current state. This is inspired by the idea of critical path in topological sort: the delay of instructions in the critical path is liable to cause higher penalties. [20]

ModelResImprove_2 The definition of TOMASULO system

```
System TOMASULO (R, N, K, U: ℕ+, FUNL : array [1..3] of ℕ+) is
 import R/R_PROGRAM(R, N)
 type
  ...
  RS_TYPE = [..., sonNum: {0} ∪ ℕ+, earliest : {0} ∪ ℕ+, latest : {0} ∪ ℕ+,
                  needChangeE : boolean, needChangeL : boolean];
 variables
   …
  RS: array[1..3, 1..U] of RS_TYPE  init  <..., 0, 0, 0, false, false >;
  maxTime: {0} ∪ ℕ+  init 0;
  ...
end system
```

In the *RS_TYPE*, we also add two fields, *needChangeE* and *needChangeL*, to notify when the *earliest* and *latest* fields should be updated. This is set by its predecessors and descendants. Besides, a field *sonNum* is used to record how many instructions are depended on this instruction. Finally, we use a global variable, *maxTime*, to store the predicted least finish time. At the end of the program execution, the predicted time equals the actual execution time.

ModelResImprove_2_2_1 The definition of ISSUE subsystem

```
system ISSUE(t:{0}∪ℕ⁺, FU:[1..3],S:ℕ⁺)is
 definitions
  ...
  findEarliestTime():ℕ⁺ = max[src(winHead,1).time, src(winHead,2).time,instruWin[winHead].time]
                          + duration(instruWin[winHead].instru.op) + transmitTime;
  findLatestTime():ℕ⁺ = max[findEarliestTime(),maxTime];
 behavior
  if ¬RS[FU,S].busy ∧ issueReady then
       RS[FU,S]:=< true, false,instruWin[winHead].time,instruWin[winHead].prog,
                   instruWin[winHead].instru.op,instruWin[winHead].instru.target,
                   src(winHead,1),src(winHead,2),0,findEarliestTime(),findLatestTime(),flase,false>;
       regFile[instruWin[nextWin].target]:=< true,<FU,S>, findEarliestTime() >;
       ||(j=1,2) if RS[FU,s].src[j].busy then
            RS[RS[FU,s].src[j].tag].sonNum++;
          end if
  end if
end system
```

At the issue point, those new fields will be initialized. Two auxiliary functions are used to calculate the expected earliest time and latest time. The earliest finish time is to be predicted as the earliest predicted ready time plus the time needed to do the calculation and transmission. Actually, it is computed along the data flow in the topological sort graph. On the other hand, the latest finish time is computed opposite to the data flow direction. An instruction's latest finish time is the minimum of each descendant's finish time minus the same descendant's operation time. As an instruction doesn't have any descendents at the issue point, its latest finish time is simply decided by the maximum of its own earliest time and the expected earliest program finish time. Meanwhile, the field of *sonNum* of its depended instruction is incremented by one. And its target register file item also writes down the predicted earliest time for its descendents.

Besides issuing, there is a particular subsystem to update the *earliest* and *latest* field for each instruction. In four conditions we need to do modification. Two of them are triggered by the field *needChangeL* and *needChangeE*. The third happens when an instruction's *latest* field is increased so much that it is larger than the global variable, *maxTime*. Then not only *maxTime* but also all those instructions whose *latest* fields used to be equal to *maxTime* need to change following *maxTime*. Finally, if an instruction has already been ready but isn't executed, its earliest finish time will be put off in every cycle. The two prediction values are changed whenever under the above four situations. After their own changes, they will go on to trigger their precedents or decedents.

ModelResImprove_2_2_5 The definition of UPDATEPREDICTION subsystem

System UPDATEPREDICTION $(t:\{0\}\cup\mathbb{N}^+, FU{:}[1..3], s{:}\mathbb{N}^+)$ *is*

Behavior

if $RS[FU,s].needChangeE$ then

$RS[FU,s].needChangeE := \text{false};$

$$RS[FU,s].earliest := \max\begin{bmatrix} RS\left[RS[FU,s].src[1].tag\right].earliest + \text{duration}(RS\left[RS[FU,s].src[1].tag\right].op) + \text{transmitTime}, \\ RS\left[RS[FU,s].src[2].tag\right].earliest + \text{duration}(RS\left[RS[FU,s].src[2].tag\right].op) + \text{transmitTime} \end{bmatrix};$$

$$\|_{FU_1=1}^{3}\ \|_{s_1=1}^{U}\ \|_{j=1}^{2}\ \text{if } RS[FU_1,s_1].src[j].busy \wedge RS[FU_1,s_1].src[j].tag =< FU,s > \text{ then}$$

$RS[FU_1,s_1].needChangeE := \text{true};$

end if

end if

if $RS[FU,s].needChangeL$ then

$RS[FU,s].needChangeL := \text{false};$

$$RS[FU,s].latest := \min\left\{\left| \begin{array}{l} RS[FU_1,s_1].latest - \text{duration}(RS[FU_1,s_1].op) - \text{transmitTime} \\ 1 \le FU_1 \le 3 \wedge 1 \le s_1 \le U \wedge RS[FU_1,s_1].busy \wedge \left(\vee_{j=1}^{2} RS[FU_1,s_1].src[j].tag =< FU,s >\right) \end{array} \right.\right\};$$

$$\|_{j=1}^{2}\ \text{if } RS[FU,s].src[j].busy$$

$RS\left[RS[FU,s].src[j].tag\right].needChangeL := \text{true};$

end if

end if

if enabled$(FU,s) \wedge \neg RS[FU,s].exec$ then

$RS[FU,s].earliest := RS[FU,s].earliest + 1;$

$RS[FU,s].latest := \max[RS[FU,s].earliest, RS[FU,s].latest];$

$$\|_{FU_1=1}^{3}\ \|_{s_1=1}^{U}\ \|_{j=1}^{2}\ \text{if } RS[FU_1,s_1].src[j].busy \wedge RS[FU_1,s_1].src[j].tag =< FU,s > \text{ then}$$

$RS[FU_1,s_1].needChangeE := \text{true};$

end if

end if

if $RS[FU,s].latest > maxTime$ then

$$\|_{FU_1=1}^{3}\ \|_{s_1=1}^{U}\ \text{if } RS[FU_1,s_1].busy \wedge RS[FU_1,s_1].latest = maxTime \text{ then}$$

$RS[FU_1,s_1].needChangeL := \text{true};$

end if

$maxTime := RS[FU,s].latest;$

end if

end system

ModelResImprove_2_2_3 The definition of EXEC subsystem

System EXEC$(t:\{0\}\cup\mathbb{N}^+, FU:[1..3])$ *is*

definitions

...

$\text{earliestReadyTime}(s:[1..U]):\{0\}\cup\mathbb{N}^+ = RS[FU,s].earliest - \text{duration}(RS[FU,s].op) - \text{transmitTime};$

$$\text{nextFuncTime}:\{0\}\cup\mathbb{N}^+ = \min_{u=1}^{FUNC[FU]}\{funcUnit[FU,u].timeStamp + \text{duration}(FU)\};$$

$$\text{appropriate}(e:[1..U]):\text{boolean} = \wedge_{s=1}^{U}\left(\begin{array}{l} \text{earliestReadyTime}(s) < \text{nextFuncTime} \\ \wedge\left(\begin{array}{l} \left(\max\left[RS[FU,s].earliest, \text{nextFunTime}\right] + \text{duration}(FU) - RS[FU,e].latest\right) * RS[FU,e].sonNum \\ > \left(\max\left[RS[FU,e].earliest, \text{nextFunTime}\right] + \text{duration}(FU) - RS[FU,s].latest\right) * RS[FU,s].sonNum > 0 \\ \vee\left(\max\left[RS[FU,e].earliest, \text{nextFunTime}\right] + \text{duration}(FU) - RS[FU,s].latest\right) < 0 \end{array}\right) \end{array}\right)$$

behavior

if enabled(e) $\wedge$ appropriate(e) $\wedge \neg funcUnit[FU, \text{freeFunc}].busy$ then

...

endif

end system

The information in *earliest*, *latest* and *sonNum* is used in deciding whether to execute an ready instruction, say e, when an idle functional unit is available. It firstly figures out when the next functional unit is available, and then finds out all of the instructions which will be ready before that. After that it computes how much the cost is to each of those functions if e is executed and the instructions need to wait. Also, it computes the cost to e itself if it waits and leaves the chance to those functions. For each instruction which cannot be executed instantly after ready, the penalty is at least the time it needs to wait times the number of its direct decedents. We use multiplication here because the delay to an instruction will in turn brings delay to each of its decedents.
Finally, we rewrite FUNC system after adding the new subsystem UPDATEPREDICT.
ModelResImprove_2_2 The definition of FUNC system:

$$System\ \ \mathrm{FUNC}(t:\{0\}\cup\mathbb{N}^{+}, FU:[1..3])\ is$$
$$behavior$$
$$\left(\underset{S=1}{\overset{U}{\|}}\ \mathrm{ISSUE}[t,FU,S]\right)\|\left(\underset{S=1}{\overset{U}{\|}}\ \mathrm{SNOOPER}[t,FU,S]\right)\|\ \mathrm{EXEC}[t,FU]\|\left(\underset{S=1}{\overset{U}{\|}}\ \mathrm{BROADCAST}[t,FU,S]\right)\|\left(\underset{S=1}{\overset{U}{\|}}\ \mathrm{UPDATEPREDICT}[t,FU,S]\right);$$
$$end\ \ system$$

The improvement takes advantage of the afterward dependence information. In a lot of cases, it can make wise choices. Using the first example in 3.1.1, we can show how well it works. At the beginning of the execution, the *earliest* and *latest* fields of each instruction are computed and can be shown as **Figure 6**. The 2nd instruction is ready, but if it is executed, the next available time for a multiplier is cycle 20. Before that the 3rd instruction will also be ready. If we choose to execute instruction 2, then according to the EXEC subsystem model, the penalty to instruction 3 is (20+20-22)*2=36. If we choose to wait and execute instruction 3, then the penalty to instruction 2 is (22+20-24)*1=18. Thus we'll intelligently choose to wait for 2 cycles and save the functional unit for instruction 3.

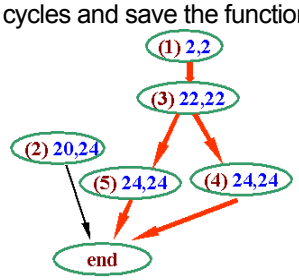


Figure 6

However, as task assignment is a NP-Complete problem, the improved algorithm also cannot always produce the best execution stream. We use least penalty for comparison. When the information cheats the system on what the penalty really is, the algorithm fails. What's more, the improved algorithm introduces great complexity and huge overhead in predicting and deciding. As stated in [], due to the limited parallelism in programs, scheduling discipline cannot largely impact performance. With the development of superscalar and SMT technology, there may be more parallelism to be exploited. A good judgment on the value of this improved algorithm can be made in two steps. The first is to test benchmarks and see whether the improvement can reduce clock cycles to a great extent; the second step is to take overheads into consideration and see whether we can really buy better performance.

8. Improved Tomasulo's Algorithm with Load/Store.

From the discussion in last two sections, we see that hardware resources play a critical role on deciding how wise the algorithm is. Besides this, there is another factor ruining its optimality – Load/Store instructions. When load and store are involved, things become more complex. We need to face up great changes both in underlying architecture and in execution time hypothesis. One concern is that the instruction execution time becomes unpredictable because of data cache misses.

Another is that WAW and WAR data hazards, which are successfully removed by register renaming, show up again in memory accesses.

There are three types of data dependence when accessing the same address:

- All loads are dependent on the previous store.
- A store is dependent on the previous store.
- A store is dependent on all loads between it and the previous store.

The only part that could be executed out of order is those load instructions between two stores surrounding them. As a result, before a load goes into the load buffer, it must check the store buffer to make sure that no active store instructions share the same address. More strictly, before a store goes into the store buffer, it must scan both store buffer and load buffer to check name dependence. If some name dependence is detected, the load or store instruction will be held in the instruction window, until the dependence was resolved and all of the previous conflicting memory accesses were moved out of the load/store buffer.

If the reason for the failure of Tomasulo's Algorithm in face of resources shortage is its impatience and dullness, then here it comes from its conservation. In the above pattern, a stalled memory access instruction will greatly damage efficiency. Due to the in-order issue principle, all of the subsequent instructions have to wait even though they are not dependent on the stalled load/store instruction.

Figure 7 shows a counterexample. Assume a cache miss happens and the store takes 50 cycles. Unfortunately, the following load instruction happens to use the same address thus has to be blocked. Consequently, the MUL instruction is also blocked although it has no dependence relationship with the previous two instructions. If it can be issued without waiting, it will be executed parallely with store instruction and the whole program can be finished in 52 cycles, saving 20 cycles.

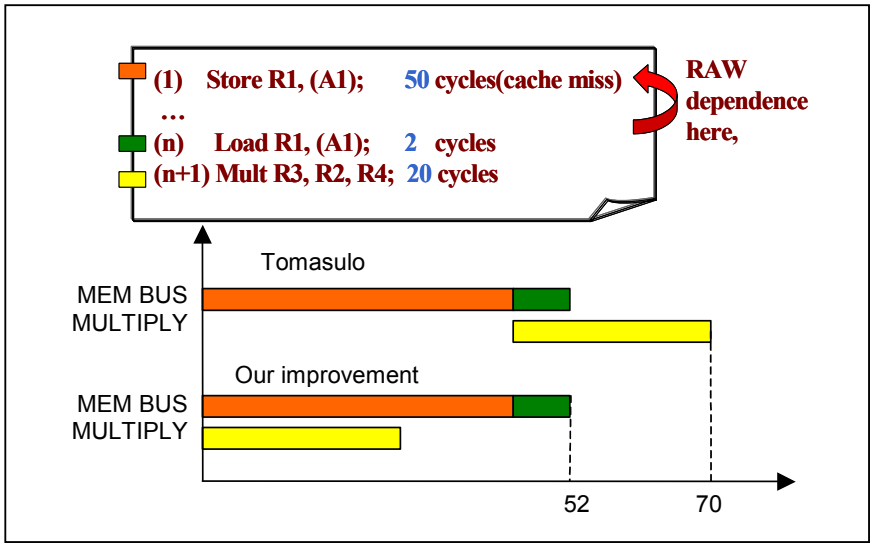
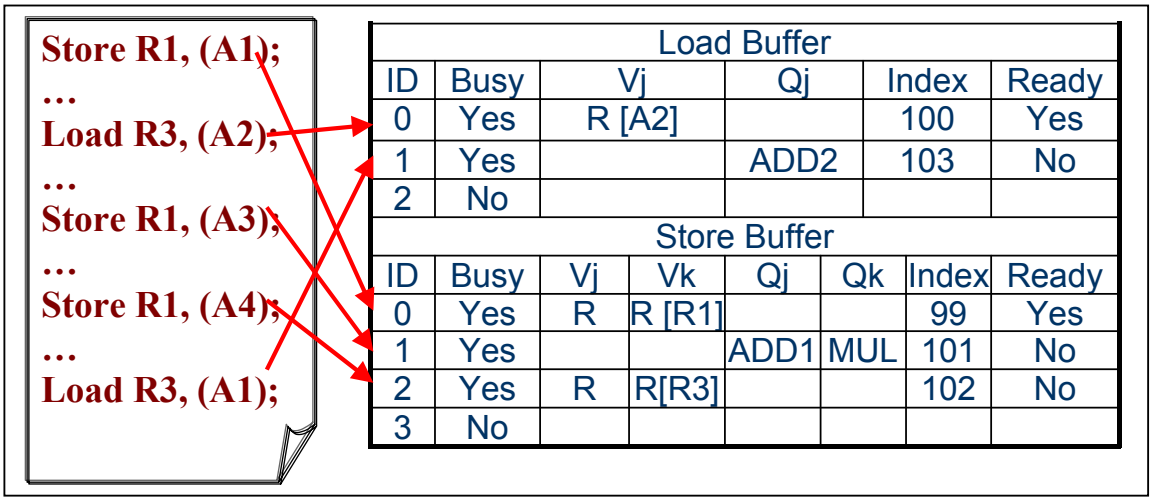


Figure 7

To solve this problem, we learn from RS scheme and add new fields to load and store buffers to record data dependence information. Also, we use instruction index number in the program to decide the order and dependence. In the improved algorithm, before a load is dispatched, all store buffers with lower instruction index are scanned to ensure no address confliction. For store instructions, the same examinations should cover both load buffer and store buffer. In this method, load and store instructions would not block the whole instruction window. **Figure 8** is an example to show the contents in the load and store buffers after issuing a series of memory access instructions.

Again, before giving the formal description of our improvement, we first build up the model for original Tomasulo's Algorithm. To concentrate on the impact of the Load/Store mechanism itself, this model is based on the initial model. Also, we assume the resources for memory accesses are infinite: infinite load and store buffer, infinite memory bus and infinite ports of main memory. It allows to simultaneously execute arbitrary number of memory access instructions and avoids the problems brought by competition for limited resources. Furthermore, to simplify our discussion, we presuppose that memory address can always be decided in decoding stage. This assumption can be removed



Load Buffer

ID	Busy	Vj	Qj	Index	Ready
0	Yes	R [A2]		100	Yes
1	Yes		ADD2	103	No
2	No				

Store Buffer

ID	Busy	Vj	Vk	Qj	Qk	Index	Ready
0	Yes	R	R [R1]			99	Yes
1	Yes			ADD1	MUL	101	No
2	Yes	R	R[R3]			102	No
3	No						

simply by changing the scan and check method.

Figure 8

In the top level model, we introduce two variables, *LoadBuffer* and *StoreBuffer* for Load/Store instructions. Besides, we add a subsystem MEMACCESS to work independently with other subsystems and imitate memory access processes. The subsystem itself is made up of four components, LOADISSUE, STOREISSUE, LOADBROADCAST and STORESNOOP. The first two are used to issue Load/Store instructions. The third part reads memory and broadcasts it on the CDB. The fourth one writes memory after resolving data dependence.

ModelLS_2 The definition of TOMASULO system:

System TOMASULO $(R,N,L,S: \mathbb{N}^+)$ *is*
import R/R_PROGRAM(R,N)
type
...
$LB\_TYPE = [busy: \text{boolean}, time: \{0\} \cup \mathbb{N}^+, prog: [0..N], address: \{0\} \cup \mathbb{N}^+, target: TARGET];$
$SB\_TYPE = [busy: \text{boolean}, time: \{0\} \cup \mathbb{N}^+, prog: [0..N], address: \{0\} \cup \mathbb{N}^+, target: TARGET, src: R\_TYPE];$
variables
...
$top: [1..N+1],$ init 1;
LoadBuffer: array$[1..\infty]$ of *LS_TYPE* init {false, 0,0,0,0};
StoreBuffer: array$[1..\infty]$ of *LS_TYPE* init {false, 0,0,0,0, < false, < 0,0 > ,0 >};
behavior
$\text{FETCH} \,||\left(\underset{FU=1}{\overset{\infty}{||}} \text{FUNC}(FU)\right)||\left(\underset{r=1}{\overset{\infty}{||}} \text{WRITERESULT}(r)\right)||\,\text{MEMACCESS};$
...
end system

ModelLS_2_5 The definition of MEMACCESS system:

System MEMACCESS *is*
behavior
$\left(\overset{\infty}{\underset{l=1}{\|}} \text{LOADISSUE}[l]\right) \| \left(\overset{\infty}{\underset{s=1}{\|}} \text{STOREISSUE}[s]\right) \| \left(\overset{\infty}{\underset{s=1}{\|}} \text{STORESNOOPER}[s]\right) \| \left(\overset{\infty}{\underset{l=1}{\|}} \text{LOADBROADCAST}[l]\right);$
end system

ModelLS_2_5_1 The definition of LOADISSUE system:

system LOADISSUE[$l : \mathbb{N}^+$] *is*
definitions
issueReadyLoad = $instruWin$[winHead].$busy \wedge instruWin$[winHead].$instru.op$ = "load"
$\wedge (\overset{\infty}{\underset{s=1}{\wedge}} StoreBuffer[s].address \neq \text{valueof}(instruWin[\text{winHead}].instru.src[1]));$
behavior
if $\neg LoadBuffer[l].busy \wedge$ issueReadyLoad then
$LoadBuffer[l]$=<true,
$time[\text{lastMemWriter}(instruWin[\text{winHead}].prog - 1, \text{valueof}(instruWin[\text{winHead}].instru.src[1]))$,
$instruWin$[winHead].$prog$,
valueof($instruWin$[winHead].$instru.src$[1])
$instruWin$[winHead].$target$ >
$regFile[instruWin[\text{winHead}].target] :=< true, < 0, l >, 0 >;$
end if
end system

The LOADISSUE works in the same way as ISSUE subsystem. It finds the earliest unissued instruction. If that's a Load instruction and doesn't have any memory conflicts with early Store instructions, it is put into the load buffer and the target register item is updated. The STOREISSUE works in the same way. The only difference is that before issuing a Store instruction, both the load buffer and store buffer need to be examined to avoid memory dependences. Here we use three assisting functions which are not a part of the TOMASULO system. One is valueof, which can calculate the address before issue stage. Another two are lastMemWriter and lastMemReader, which tells us the accurate time of the last write or read process of the same memory. If there is no previous Load/Store, then the function returns 0.

ModelLS_2_5_2 The definition of STOREISSUE system:

```
system STOREISSUE[s : ℕ+] is
  definitions
    issueReadyStore = instruWin[winHead].busy ∧ instruWin[winHead].instru.op = "store"
                      ∧ (∧_{s=1}^{∞} StoreBuffer[s].address ≠ valueof(instruWin[winHead].instru.src[1]));
                      ∧ (∧_{s=1}^{∞} LoadBuffer[s].address ≠ valueof(instruWin[winHead].instru.src[1]));
  behavior
    if ¬StoreBuffer[s].busy ∧ issueReadyStore then
      StoreBuffer[s]=<true,
                      max[ time[lastMemWriter(instruWin[winHead].prog − 1, valueof(instruWin[winHead].instru.src[1])),
                           time[lastMemReader(instruWin[winHead].prog − 1, valueof(instruWin[winHead].instru.src[1])) ],
                      instruWin[winHead].prog,
                      valueof(instruWin[winHead].instru.src[1])
                      src(winHead, 2) >
    end if
end system
```

ModelLS_2_5_4 The definition of STORESNOOPER system:

```
System STORESNOOPER(s : ℕ+) is
  behavior
    ||_{s=1}^{S} ||_{bus=1}^{∞} if StoreBuffer[s].busy ∧ StoreBuffer[s].src.busy ∧ CDB[bus].busy
                ∧ StoreBuffer[s].src.tag = CDB[bus].tag then
                  StoreBuffer[s].src.time := CDB[bus].time;
                  StoreBuffer[s].src.busy := false;
              end if
    if ¬StoreBuffer[s].src.busy then
      time[StoreBuffer[s].prog] := max[StoreBuffer[s].time, StoreBuffer[s].src.time] + memAccessTime;
      Completed[StoreBuffer[s]].prog] := true;
      StoreBuffer[s].busy := false;
    endif
end system
```

STORESNOOPER has the same mechanism with the SNOOPER subsystem of RS. It monitors CDB and gets the data with the same tag as its source tag. After that the data is transmitted to the memory and the instruction finishes. The finish time of a Store instruction should be the time when it has been issued and also gets the operand, plus the time needed to access memory. Due to the possibility of cache miss and page fault, the access time is not fixed and we just use an oracle function memAccessTime to describe it.

ModelLS_2_5_4 The definition of LOADBROADCAST system:

```
System LOADBROADCAST(l : ℕ+) is
  behavior
    ||_{bus=1}^{∞} if LoadBuffer[l].busy ∧ ¬CDB[bus].busy then
          CDB[bus].busy := true;
          CDB[bus].tag: =< 0,l >;
          CDB[bus].time: = LoadBuffer[l].time + memAccessTime;
          time[LoadBuffer[l].prog] := LoadBuffer[l].time + memAccessTime;
          Completed[LoadBuffer[l].prog] := true;
          LoadBuffer[l].busy : = false;
        end if
end svstem
```

For the Load instruction, as memory address is known at the issue point, it can be executed as soon as it is issued. Then the received data is broadcast by CDB and the instruction is marked as finished. Here, the finish time is simply the issue time plus the memory access time.

Finally, we give the improved algorithm based on the above model. One modification is made in ISSUE systems. A Load/Store instruction is issued without checking dependences. However, before it is dispatched and the memory is accessed, name dependence must be checked first to guarantee right results. This is achieved by comparison with the memory addresses of all those load and store buffers having lower instruction indices.

ModelLSImproved_2_5_1 The definition of LOADISSUE system:

```
system LOADISSUE[l : ℕ+] is
  definitions
    issueReadyLoad = instruWin[winHead].busy ∧ instruWin[winHead].instrı
  ...
end system
```

ModelLSImproved_2_5_2 The definition of STOREISSUE system:

```
system STOREISSUE[s : ℕ+] is
  definitions
    issueReadyStore = instruWin[winHead].busy ∧ instruWin[winHead].instru.o
  ...
end system
```

ModelLSImproved_2_5_3 The definition of STORESNOOPER system:

System STORESNOOPER($s:\mathbb{N}^+$) *is*
behavior
...
if $\neg StoreBuffer[s].src.busy$
$\wedge \overset{\infty}{\underset{ps=1}{\wedge}} (StoreBuffer[ps].prog < StoreBuffer[s].prog \wedge StoreBuffer[ps].address \neq StoreBuffer[s].address)$
$\wedge \overset{\infty}{\underset{ps=1}{\wedge}} (LoadBuffer[ps].prog < StoreBuffer[s].prog \wedge LoadBuffer[ps].address \neq StoreBuffer[s].address)$ then
...
end if
end system

ModelLSImproved_2_5_4 The definition of LOADBROADCAST system:

System LOADBROADCAST($l:\mathbb{N}^+$) is
behavior
$\overset{\infty}{\underset{bus=1}{\|}}$ if $LoadBuffer[l].ready \wedge \neg CDB[\text{bus}].busy$
$\wedge \left(\overset{\infty}{\underset{ps=1}{\wedge}} LoadBuffer[ps].prog < LoadBuffer[s].prog \wedge LoadBuffer[ps].address \neq LoadBuffer[s].address \right)$ then
...
end if
end system

Because different memory access orders may result in different cache misses, so it's not meaningful to discuss algorithm optimality here. However, intuitively our improvement can generally beat the original algorithm as it removes unnecessary stalls and tries to exploit parallelism to the greatest extend. A more aggressive improvement can be made by adding data forwarding between loads and stores.[21,22] Due to its complexity, we don't discuss the details here.

9. Conclusion

This report examines whether Tomasulo's algorithm is optimal in a formal way. Mathematical models are built up for the behavior of Tomasulo's algorithm as well as a general data-driven system which describes the conduct of all kinds of dynamic scheduling mechanisms.

We give the formal proof that under certain assumptions, including that we have infinite functional units, infinite bandwidth and all of the instructions can be issued instantly after they enter the instruction window, no other dynamic scheduling algorithms can give a better performance than Tomasulo's algorithm. The main reason is that when we have boundless resources, the key problem is to keep the resources as busy as possible. The transmission delay doesn't imperil its optimality. Even when the instruction window buffer size is limited, Tomasulo's algorithm works well because window buffer is essentially a queue and no tricky choices can be made when fetching and replacing instructions.

Nonetheless, Tomasulo's Algorithm does not win in face of realistic hardware restrictions. In this condition, dynamic scheduling is fundamentally a task arrangement problem which is NP-hard. So it's

impossible to find out a hardware algorithm to solve it in polynomial time. Tomasulo's algorithm uses greedy mechanism to give an approximate solution. However, as it doesn't consider afterward dependences, which can be important information in scheduling decision, it is not the best algorithm and can be improved.

Moreover, when we take Load/Store instructions and issue delays into consideration, the algorithm is not optimal, either. The reason for the first case is its conservation and can be improved by making it more aggressive. In the second case, as the algorithm is constrained by a local view, there is no room for improvement except boosting instructions issue speed.

Besides giving out counter-examples, we also present two improved algorithms under limited resources and Load/Store instructions respectively. The former algorithm is much more complex and cannot improve performance dramatically given realistic programs. The latter one is reasonable, although still means more overhead and complexity.

As a conclusion, future work should be focused on choosing parameters that work well together, not in inventing new techniques. Architects hit the target by careful, quantitative analysis. Questions that need to be answered include: how large RS should be, how many functional units should be used, how to enhance CDB bandwidth, how to issue maximum instructions in one cycle, and how to reduce transmission delay by directly forwarding results inside functional units. While new techniques such as SMT bring more instruction level parallelism, how to exploit the parallelism will continue to be a hot topic in future microarchitecture. However, as the problem itself is NP-hard, there is little chance that the milestone lies in finding an optimal algorithm.

Reference

[1] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development,* 11(1):25-33,1967

[2] W. Damm and A. Pnueli. Verifying Out-of-Order Executions. *CHARME'97*:23-47, Chapman&Hall, 1997

[3] T. Arons and A. Pnueli. Verifying Tomasulo's Algorithm by Refinement. *Technical report, Dept. of Comp. Sci., Weizmann Institute*, Oct 1998

[4] K.L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. *CAV'98*:110-121,1998

[5] J.U. Skakkebæk, R.B. Jones, and D.L. Dill. Formal Verification of Out-of-Order Execution Using Incremental Flushing. *CAV'98*:98-110,1998

[6] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. A Proof of Correctness of a Processor Implementing Tomasulo's Algorithm without a Reorder Buffer, *CHARME*, 1999.

[7] R.M. Hosabettu. Systematic Verification of Pipelined Microprocessors. *Ph.D. dissertation*.

[8] M. Butler and Y. Patt. An Investigation of the Performance of Various Dynamic Scheduling Techniques, *Proc.ISCA-92*:1-9.

[9] W.K. Norton, Pooling resources in Tomasulo algorithm *http://citeseer.nj.nec.com/165782.html.*

[10] V. Van Dongen, G.R. Gao, and Q. Ning. A polynomial time method for optimal software pipelining. *Proc. of the Conference on Vector and Parallel Processing, CONPAR-92*:613-624, Sept 1992. Also in *LNCS-634*.

[11] D. Bernstein, M. Rodeh, I. Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. *IEEE Trans. on computers,*38(9),1989

[12] J. Blazewicz, J.K. Lenstra and A.H.G. Rinnoy Kan. Scheduling subject to resource constraints: classification and complexity, *Discrete Applied Mathematics*,5:11-24, 1983.

[13] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287--326, 1979.

[14] U. Feige and C. Scheideler Improved Bounds for Acyclic Job Shop Scheduling. May,1998

[15] W. Zuckerman. NP-complete problems have a version that's hard to approximate. *Proc. Eight Ann. Structure in Complexity Theory Conf., IEEE Computer Society*:305-312, 1993.

[16] D. Bernstein, M. Rodeh, and I. Gertner. Approximation algorithms for scheduling arithmetic expressions on pipelined machines. *J. Algorithms* 10:120-139, 1989.
[17] S. Rao, and A.W. Richa. New approximation techniques for some ordering problems. *Proc. 9th Ann. ACM-SIAM Symp. on Discrete Algorithms, ACM-SIAM*:211-218, 1998.
[18] M.X. Goemans, M. Queyranne, A.S. Schulz, M. Skutella, and Y. Wang. Single machine scheduling with release dates. *Unpublished manuscript*, 1998
[19] M. Skutella. Approximation algorithms for the discrete time-cost tradeoff problem. *Proc. 8th Ann. ACM-SIAM Symp. on Discrete Algorithms, ACM*-SIAM:501-508, 1997.
[20] M.A. Weiss. Data Structures & algorithm Analysis in C++ (2nd edition). *Addison-Wesley*:330-332, 1999.
[21] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel. The Microarchitecture of the Pentium 4 Processor.
[22] R.E. Kessler. The ALPHA21264 Microprocessor. 24-36, 1999.
[23] http://www.nada.kth.se/~viggo/problemlist/compendium.html