CSE548: Optimality of Tomasulo's Algorithm

Amol Prakash, Sumit Sanghai

March 16, 2002

1 Introduction

The focus of architects has always been on faster execution of instructions. After pipelining, the architects realised that the processor resources were still not being fully exploited. By doing an out-of-order execution they could perform better. R.M. Tomasulo [10] in 1967 gave an algorithm which improved the performance by exploiting Instruction Level Parallelism. It introduced two interesting concepts: Common Data Bus and Virtual Register Renaming.

The Tomasulo algorithm is the classical scheduler supporting out-of-order execution. It is widely used in current high performance microprocessors. The correctness of this algorithm has been proved in a variety of cases. Nearly all the processors these days are based on this algorithm. Even though lots of changes have happened in processors since 1967, but still Tomasulo's algorithm forms the basis of scheduling instructions in all of them. Cycle time has reduced drastically. Hardware has become a lot cheaper. Communication delays are becoming bottlenecks. Even with all these developments, processors are still based on Tomasulo's algorithm.

As of today, we do not know whether Tomasulo's approach is best for the present state-of-the-art processors and those belonging to the future. In this project we aim to figure this out. First we try and figure out the assumptions that we need to make to prove the algorithm optimal. By optimality we mean that under certain hardware assumptions, a program (which follows certain rules) would take the minimum number of cycles. After we have proven the optimality, we will analyse the assumptions which fail in today's processors. Then we will give worst case bounds on the performance loss if this algorithm is employed in them.

In Section 2, we discuss the variety of cases where Tomasulo's algorithm has been proved correct. This will help us analyse the approach that one follows while studying a hardware algorithm. In section 3, we figure out the assumptions that we need to prove Tomasulo's algorithm optimal. In its subsections we state the assumption and find the worst case bound on the algorithm's performace if this assumption fails. Also we prove that the worst case analysis is indeed worst case. In Section 4, we discuss the proof of optimality of Tomasulo's algorithm given the assumptions. In Section 5 and 6, we discuss the performance of Tomasulo's algorithm under in-order and out-of-order issue and dispatch. In Section 7 , we state our conclusions.

2 Related Work

In recent years, much of the work related to Tomasulo's algorithm has been focussed on proving its correctness. Different approaches have been used to prove the correctness of Tomasulo's algorithm.

Damm and Pnueli [2] give a proof based on refinement. This proof is in two stages, first refining a sequential specification to an intermediate model based on partially ordered executions,

and then refining this model to the implementation. Arons and Pnueli [1] extend this approach by using the concept of "predicted value". The "predicted value" is an auxillary variable which helps in comparing the implementation against its specification without constructing an intermediate abstraction. They verify the proof using the PVS thorem prover. The proof works for arbitrary configurations of unlimited size and is independent of the operations appearing in the instructions. But, the model is limited to non-branching programs. Also, the proof does not deal with loads and stores.

Mcmillan [5] gives a model checking based verification of Tomasulo's algorithm. He partly automates the proof of Damn and Pnueli. He refines the specification directly to implementation with no intermediate step by using compositional model checking. The proof is fully verified by the SMV verifier. Mcmillan uses symmetry reductions to manually decompose the proof into lemmas about small collections of state components. The refinement maps that he uses also seem a fairly natural representation of the function of the various machine components. The proof can also be possibly reused to verify other architectures. The problem here is that the proof is dependent on the actual configuration and the arithmetic functions used.

Skakkebaek, Jones and Dill [4] verify a complex model of out-of-order execution using the Stanford Validity Checker. They provide a formal method called incremental flushing to show that the intermediate abstraction is functionally equivalent to the specification machine. The approach used by Arons and Pnueli, as discussed above, use refinement on the specification level, thus obviating the need for a second stage.

All of the proofs discussed above don't deal with exceptions and the liveness properties.

Sawada and Hunt [9] provide a verification of a processor implementing Tomasulo's algorithm with a reorder buffer, exceptions and speculative execution. They use a table of history variables, called MAETT. The MAETT is an intermediate abstraction that contains selected parts of the implementation as well as extra history variables and variables holding abstract values. A predicate relating the MAETT and the implementation is found by manual inspection and proven by induction to be an invariant on the execution of the implementation. The problem here is that a lot of manual effort is required to construct the intermediate abstraction.

Hosabettu, Gopalakrishnan and Srivas [8] use the completion functions approach to prove the correctness of Tomasulo's algorithm. They also give a proof for a processor implementing Tomasulo's algorithm without a reorder buffer [6]. This proof also relies on the completion functions approach which they introduced in [7]. The verification requires the user to manually define a set of completion functions, one per unfinished instruction in flight, describing how that instruction will be completed, given that all the instructions on which it has a data dependency have already been completed. This effectively results in a manual recreation of much of the hardware functionality. Furthermore, the user has to manually define a way to compose these completion functions in order to form the abstraction function for the processor.

Kroenig, Mueller and Paul [3] give a mathematical proof of a Tomasulo scheduler supporting precise interrupts. They show that the data consistency is maintained and that the algorithm is deadlock-free and fair. The fairness and the deadlock-free execution are shown by a worst case runtime analysis of an arbitrary program. The proof, however, omits the scheduling problem within complex function unit such as iterative floating point dividers. Also, the proof lacks automatization.

3 Assumptions Required

As discussed in previous sections, Tomasulo's algorithm has been proved correcting a variety of scenarios. We are trying to prove its optimality under a given set of assumptions. There are two steps in doing this. The first is to figure out the assumptions that we require. The next step is to prove it optimal given those set of assumptions.

The assumption set should be necessary and sufficient. This means that if any assumption fails, then we should be able to prove that the algorithm is not optimal. The easiest approach would be a proof by evidence, i.e. find a sequence of instructions for which the algorithm does not perform the best if that assumption failed. We have also done a worst case analysis and found the performance loss in case the assumption failed. To show that the assumption set to be sufficient we prove Tomasulo to be optimal given the set of assumptions.

We initially assume that Tomasulo's algorithm follows an in-order issue and an in-order dispatch, and so does any other algorithm if we are comparing Tomasulo against anything. Also for sake of simplicty, we assume that there are no branch instructions or load-store instructions.

In the following subsections we form the assumption set by adding assumptions one by one.

3.1 Infinite Reservation Stations

In Tomasulo's algorithm, after an instruction is issued, it is put into the reservation station. It waits here till its operands are available. This could be because of a dependency on a previous instruction (which is executing or waiting). After both of its operands are available it is executed on the corresponding functional unit (if there is one free). If we assume enough functional units, then as soon as the operands become ready, the instruction is executed. For simplicity purposes, let us assume that there only be one type of instructions : DIV. The analysis gets a lot more complex if we assume a variety of instruction types, but the result is still the same. Why did we choose DIV instruction particularly ? Again, just to make the analysis simpler and more convincing.

3.1.1 Why Infinite?

Having infinite reservation stations intuitively seems as a requirement to achieve best performance, so let us prove that we need to make this assumption in order to prove Tomasulo's algorithm optimal. Suppose that instead we do not have infinite reservation stations. Let there be m reservation stations in the processor. Suppose there is a sequence of *(m+2)* instructions :
DIV R1, R2, R3
DIV R4, R1, R5
DIV R6, R1, R7
DIV R8, R1, R9
.
.
.
DIV Ri, R1, Rj

All instructions (leaving the first one) are dependent on the first one. As the first instruction is dispatched (from the reservation station), the second instruction enters the reservation station. But it cannot be executed because it needs the result of the first instruction. Similarly, the rest instructions keep getting issued and then wait in the reservation station because they are all dependent on the first instruction. This happens till the reservation station gets full. So m cylces after

the dispatch of the first instruction, the reservation stations get full. Assuming that division takes more than m cycles to execute, after m cycles there are more instructions that were dependent only on the first instruction and have still not got issued because the reservation stations are full. So, when the first instruction finishes execution there are m DIV instructions waiting in the reservation station. Now as soon as the result for *R1* is computed, there are m DIV instructions which can be dispatched simultaneously. But more could have been dispatched, had there been space in the reservation stations in the first case. But since it is finite, only m instructions get executed.

The above analysis shows a case of performace loss if there are finite number of reservation stations. So Tomasulo's algorithm will not perform optimally in the absense of infinite reservation stations. In the following paragraphs, we do a worst case analysis of the above problem.

3.1.2 Worst Case Performance Loss

The case we present here may or may not be the worst case but it exhibits a significant performance loss. Consider n_1 instructions, each of which is dependent on the previous instruction, followed by n_2 mutually independent instructions, each of which is dependent on the last instruction from the first set. Assume that all instructions are of the same type each taking k cycles. Let $n_2 = n_1 * (k-1)$. When there are infinite reservation stations, the total execution time would be $n_1 * k + k$ cycles. This is because when the first n_1 instructions finish executing, all the remaining n_2 instructions would have entered the reservation station and all of them can be dispatched simultaneously. Assume that the number of functional units is greater than n_2.

When the number of reservation stations is finite, say m, after the first n_1 instructions are executed, only m instructions would have been present in the reservation station. Thus it would take $n_2 - m$ additional cycles to bring the instructions into the reservation station. Thus the total number of cycles taken is equal to $n_2 - m + n_1 * k + k$. After substituting for n_2 and taking n_1 sufficiently high, we get that there is a scaleup of 2.

3.2 Infinite Functional Units

Functional units are required in any hardware to compute functions like addition, multiplication, division etc. Tomasulo's algorithm uses reservation stations to prepare operands for these functional units. It exploits Instruction Level Parallelism by implementing out-of-order execution. So the instructions which have a dependency wait in the reservation stations till the operand becomes available after which they are executed (in functional units). For simplicity sake we assume only one type of instructions : DIV. The analysis gets a lot more complex if we assume a variety of instruction types, but the result is still the same. Why did we choose DIV instruction particularly ? Again, just to make the analysis simpler and more convincing.

3.2.1 Why Infinite?

Having infinite hardware is a dream, but we need to make this assumption in order to prove Tomasulo's algorithm optimal. Suppose that instead we do not have infinite DIV units. Let there be m DIV units in the processor. Suppose there is a sequence of *(m+2)* instructions :
DIV R1, R2, R3
DIV R4, R1, R5
DIV R6, R1, R7
DIV R8, R1, R9
.

.
.
DIV Ri, R1, Rj

All instructions (leaving the first one) are dependent of the first one. As the first instruction is dispatched, the second instruction enters the reservation station. But it cannot be executed because it needs the result of the first instruction. Similarly, the rest m instructions wait in the reservation station because they all are dependent of the first instruction. Assume that there is enough space in the reservation station and division takes more than m cycles to execute. So, when the first instruction finishes execution there are at least *(m+1)* DIV instructions waiting in the reservation station. Now as soon as the result for *R1* is computed, there are at least *(m+1)* DIV instructions which can be dispatched. But at this time there are only m available DIV units. So at least one instruction would have to wait till it gets a free DIV unit.

The above analysis shows a case of performace loss if there are finite number of functional units, under certain assumptions. So Tomasulo's algorithm might not perform optimally in the absense of infinite hardware. In the following paragraphs, we do a worst case analysis of the above problem.

3.2.2 Worst Case Performace Loss

We assume that the number of functional units, m, is less than the maximum of the number of cycles taken by all the instructions, denoted by k. This is especially true for the DIVISION functional units.

Again, the following may or may not be the worst case.

Consider n independent instructions each of which take k cycles to execute. It is easy to see that when the number of functional units is infinite, the total number of cycles required to execute these instructions is equal to $n + k$. It takes 1 cycle to fetch each instruction into the reservation station. As soon as any instruction enters the reservation station it gets scheduled into some functional unit. This is because it is independent of all the previous instructions and also a functional unit is available.

Now, consider the case when there are m functional units available where $m < k$. Consider the $m + 1^{th}$ instruction. It would arrive in the reservation station at the $m + 1^{th}$ cycle. But, at that time all the functional units are full. This instruction would get scheduled at the $k + 1^{th}$ cycle, when the first instruction is finished executing, and would finish at the $2k + 1^{th}$ cycle. Similarly the $2m^{th}$ instruction would finish at the $2k + m^{th}$ cycle. By repeating this argument we get that the total execution time is $(n/m) * k + m$ cycles (assuming n is a multiple of m). Thus we can see that there is a scaleup of k/m.

3.3 Global Communication

In Tomasulo's algorithm, communication between functional units, reservation stations and buffers is done with the help of Common Data Bus. This bus connects all the units which helps them to transfer data amongst themselves. If there is a contention, then that is resolved using priority. A unit which takes more delay to compute has a higher priority to contend for the Common Data Bus. So, a faster unit will have to wait to put the data on the Common Data Bus.

To contend for the bus, the unit has to request it two cycles before the output becomes available. After this a priority resolution is done, after which the bus is allotted. Bus contention can be a problem, because it potentially can lead to loss in performance. In the following example we cite a sequence of instructions where the priority would lead to a non-optimal evaluation of instructions.

Instead having a connection from every unit to every unit would lead to optimal performance. This way, contention would never be a problem.

3.3.1 The Worst Case

Again, this may or may not be the worst case but it exhibits significant performance loss. Consider a DIV instruction followed by n_1 DIV instructions, all of which depend on the first one. After this, there are n_1 ADD instructions each of which depends on the corresponding DIV instruction.

As the first instruction executes, the rest instructions wait in the reservation stations because of dependency. After the result of first instruction is available, then all the rest n_1 instructions get dispatched. Assume that there is sufficient functional units. Now the ADD instruction waits in the reservation station. Suppose DIV takes k cycles to complete. So, after k cycles, all the DIV instructions complete execution. Now since there is only one common data bus, each one of them will contend for that. It would be allotted on a first come first server basis, because all the units have the same delay. So the last instruction would be able to transmit its result only after n_1 cycles. Thus the last ADD instruction would get dispatched only $n_1 + 1$ cycles after the execution of DIV instructions.

On the other hand, had each unit been connected to each unit, this situation would not have arrived. The last ADD would have received the result immediately after the execution of the last DIV instruction. So we get a performance loss of n_1 cycles. This is proportional to the number of instructions.

4 Proof of Optimality and Sufficiency

After making a necessary assumption set, let us try and prove it to be sufficient. We do it by way of induction.

We compare the performance of Tomasulo's algorithm over an optimal algorithm using the same hardware. The hardware used by the optimal algorithm might have different structure, but it would still have similar organization i.e. it would have an instruction buffer from where instructions are fetched, a buffer (reservation station) where instructions are stored before they are dispatched, and functional units. There may be differences in the number of pipeline stages and links, but overall the organisation looks the same.

Also, we assume that the same sequence of instructions is being fed to both the algorithms and both the algorithms follow in-order issue strategy.

4.1 Variable of Induction

For every instruction, consider a number p which indicates the number of cycles remaining for it to complete execution.

1. For executed instructions p is zero, since these have finished execution.
2. For an instruction which is executing, p is number of cycles remaining for it to finish execution.
3. For a ready instruction which is in the reservation station, p is 1 greater than the number of cycles required to complete execution. The intuition for this is that the instruction will take at least one additional cycles to move from reservation station to functional unit.

4. For an instruction which is in the reservation station but still not ready, p is 2 greater than the number of cycles required to complete execution. The intuition for this is that the instruction will take at least two additional cycle to move from reservation station to functional unit, one for getting ready and second to move from reservation station to functional unit.

5. For instructions in instruction buffer, give a *line_number* to each instruction starting with 1 being given to the instruction next to be issued. Now, p is ($line\_number + 2$) greater than the number of cycles required to complete execution. The intuition for this is that the instruction will take *line_number* cycles to get issued, one cycle to resolve dependency and one additional cycle to move from reservation station to functional units. We may not need the extra cycle in which dependency is resolved (for independent instructions), but we want to distinguish between the first instruction to-be-issued and the instructions which are not-ready and waiting in the reservation stations.

After n cycles, if we are following Tomasulo's algorithm, the value of p for all instructions is less than or equal to the value of p for the corresponding instruction while using the optimal algorithm. The variable of induction is n.

4.2 Base Case

For $n = 1$, all but one instruction would be following the case 5. So for both Tomasulo's algorithm and the optimal algorithm, they would have equal value of p. The first instruction would have entered the reservation station, and since it would not be dependent on any other instruction (trivially), it would follow case 3. It would be ready to be dispatched. So again both Tomasulo's algorithm and the optimal algorithm have equal value for p.

4.3 Induction Hypothesis

Assume that the statement is true for all $k \leq n$ i.e. after every clock cycle till n cycles, using Tomasulo's algorithm, the value of p for all instructions is less than or equal to the value of p for the corresponding instruction while using the optimal algorithm.

4.4 Inductive Case

Consider the $n + 1$ cycle. Let's analyse each type of instruction individually as to its status in the nth cycle.

1. For executed instructions p was zero while using Tomasulo's algorithm. The value of p cannot be smaller than zero for the same instuctions while using the optimal algorithm.

2. For an instruction which is executing, suppose the value was $p1$ (using Tomasulo's algorithm) and $p2$ (using optimal algorithm) after n cycles. By induction hypothesis, $p1 \leq p2$. After one cycle, the value using Tomasulo's algorithm decrements by one (since the instruction is executing) which is the maximum that the value can decrement by in any case. So $p1 - 1 \leq p2 - 1$. Here we have used the assumption regarding the instant writeback of results using instant communication instead of CDB.

3. For a ready instruction which is in the reservation station, suppose the value was $p1$ (using Tomasulo's algorithm) and $p2$ (using optimal algorithm) after n cycles. By induction hypothesis, $p1 \leq p2$. After one cycle, the value using Tomasulo's algorithm decrements by

one (since the instruction is ready, so it gets dispatched) which is the maximum that the value can decrement by in any case. So $p1 - 1 \leq p2 - 1$. Here we have assumed that in Tomasulo's algorithm, we have functional units available to us for dispatch which comes from the assumption about infinite functional units.

4. For an instruction which is in the reservation station but still not ready, suppose the value was $p1$ (using Tomasulo's algorithm) and $p2$ (using optimal algorithm) after n cycles. By induction hypothesis, $p1 \leq p2$. There are three cases possible :

   - Suppose using Tomasulo's algorithm, the instruction becomes ready in $n + 1$ cycle. So after $n + 1$ cycle, the value using Tomasulo's algorithm decrements by one, so it gets dispatched which is the maximum that the value can decrement by in any case. So $p1 - 1 \leq p2 - 1$.
   - Suppose the instruction does not get ready in either case. So the values of p remain the same after $n + 1$ cycles. So, following induction hypothesis, $p1 \leq p2$.
   - Suppose the instruction remains not-ready using Tomasulo's algorithm in $n + 1$ cycle. But, it gets ready in the optimal algorithm. Now, since the instruction got ready in the $n + 1$ cycle, this means that all the instructions on which it depended finished execution then. So at the end of n cycles, all these instructions were either executing or had finished execution. So the value of p for all these instruction was either 0 or 1. By induction hypothesis, the value of p has to at most 1 for all these instructions even when we are using Tomasulo's algorithm. Since an instruction takes at least 1 cycle to execute, all these instructions should also be executing or finished execution at the end of n cycles. Again assuming instant writeback, all these instructions would finish execution at the end of $n + 1$ cycles. So it cannot be possible that this instruction remains not-ready while using Tomasulo's algorithm after $n + 1$ cycles.

5. For an instruction in instruction buffer, suppose the value was $p1$ (using Tomasulo's algorithm) and $p2$ (using optimal algorithm) after n cycles. By induction hypothesis, $p1 \leq p2$. In every cycle, Tomasulo's algorithm moves one instruction from the instruction buffer to the reservation station. So this means, after one cycle, the value using Tomasulo's algorithm decrements by one (since one instruction is issued, all $line\_numbers$ reduce by 1) which is the maximum that the value can decrement by in any case. So $p1 - 1 \leq p2 - 1$. Here we have assumed that in Tomasulo's algorithm, we have reservation stations available to us for dispatch which comes from the assumption about infinite reservation stations.

Now, by principle of mathematical induction, after n cycles, if we are following Tomasulo's algorithm, the value of p for all instructions is less than orequal to the value of p for the corresponding instruction while using the optimal algorithm. This shows that the sequence of instructions cannot execute faster for the optimal algorithm against Tomasulo's algorithm. So this shows that Tomasulo's algorithm is optimal and the set of assumptions is sufficient.

5 Out-Of-Order and In-Order Issue

In the previous sections, we have proved Tomasulo to be optimal going with in-order issue. For this we made certain assumptions only under which Tomasulo was proved optimal. On the other hand, we can also assume out-of-order issue, but this would be more of a plugin for the Tomasulo's algorithm. Reordering a sequence of instructions can potentially yield a better performance, but this

reordering would be the same for Tomasulo and any other algorithm. So what any algorithm would do is to follow in-order issue over this reordered sequence. So we do not discuss the performance of out-of-order issue.

6 Out-Of-Order and In-Order Dispatch

As for dispatch, Tomasulo's algorithm does not have any good scheduling algorithm for this. It just follows an in-order dispatch strategy. But we need to compare its performace against other algorithms which might follow different strategy.

6.1 Infinite Functional Units

In the case of infinite hardware it does not make any difference. As any instruction gets ready, it is dispatched because there is no problem regarding availability of hardware. So no instruction has to wait in the reservation station. So in-order or out-of-order dispatch will not make any difference.

6.2 Finite Functional Units : In-Order Dispatch

Uptil now, we have been assuming in-order dispatch for all our proves, where we showed Tomasulo to be non-optimal in the case of finite hardware against infinite hardware. Let us try and analyse a similar case against finite hardware, i.e. against an ideal algorithm which follows in-order dispatch but has finite number of functional units.

6.2.1 Counter Example

Consider the following sequence of 5 instructions where each instruction takes 4 cycles. Instruction 2 and 3 depend on instruction 1. Instruction 4 is independent. Instruction 5 depends on instruction 3. Also assume that there are two functional units available to us.

The way Tomasulo would execute this, instruction 1 gets executed in first cycle. In cycle 4, instruction 4 is issued so it can be dispatched (because instructions 2 and 3 are waiting). Later on, in cycle 5, instruction 2 is scheduled (when instruction 1 ends). Instruction 4 finishes execution in cycle 7, and in the 8th cycle instruction 3 is scheduled. Instruction 5 is scheduled in cycle 12th (when instruction 3 3 ends). So the sequence is completely executed by the 15th cycle.

Suppose now we had an ideal algorithm that also did an in-order dispatch. This algorithm schedules instruction 1 in the 1st cycle. Instructions 2 and 3 wait for 1 to finish. Since it is an in-order dispatch, no other instruction can get dispatched. This algorithm does not schedule the 4th instruction, but instead in the 5th cycle, both 2nd and 3rd instructions get scheduled. And then in the 9th cycle, both instructions 4 and 5 get scheduled. These finish in the 12th cycle. So the sequence is completely executed in the 12th cycle.

The above example showed that even against an algorithm following in-order dispatch, we are able to show an improvement of one cycle.

6.3 Finite Functional Units : Out-of-Order Dispatch

As we showed above, Tomasulo's algorithm is not optimal if we follow in-order dispatch over finite number of functional units, so trivially it would perform still worse against an algorithm which performs out-of-order dispatch. Here we try an give a bound on the performance loss by doing a worst case analysis.

6.3.1 Worst Case

Assuming there are m functional units (taking only one type of instruction), we can prove that the maximum performance loss that we encounter by not using out-of-order dispatch is a factor of $(2 - 1//m)$. The proof for this being the worst case performace loss is related to the proof of the famous online scheduling problem, and can be found in [11].

6.3.2 Worst Case Example

Here we give an example where we encounter a performance loss of the order given above. Assume that we have m functional units (all for one instruction which takes k cycles to execute). Consider a sequence of $m * (m - 1)$ instructions all which are independent. This is then followed by m instructions each of which depends on its previous instruction. So the instruction at the beginning of this m block depends on the last instruction in the independent instructions block. Assume that k is much larger than m (true for DIV instructions).

Tomasulo would schedule the independent instructions in blocks of m becasue of hardware constrainst and after that the dependent instructions go one by one. So the total number of cycles taken are $(m - 1) * k + m + m * k$, which is approximated to $(2m - 1) * k$.

An algorithm following out-of-order dispatch, would schedule $(m - 1)$ instruction blocks (from independent block) and 1 instruction from the dependent block (the appropriately chosen block and the dependent instruction). This way the sequence of instructions can be scheduled in $m * k$ cycles. Taking the ratio, we get that the performance loss is $(2m - 1)/m)$ which is same as the worst case claimed in [11].

7 Conclusions

In the above sections we have figured out the assumptions that we need in order to prove optimality of Tomasulo's algorithm. As we showed above, we need infinite reservation stations and functional units. Other than that we also need hardware for global communication instead of a Common Data Bus. This can be met by connecting each unit with the other.

Along with stating the assumptions, we also showed examples where there was a heavy loss in performance. Also, we have tried to bring out the worst cases, but have not given proofs showing that the cases dealt are indeed worst case. In the absense of infinite reservation stations, we showed a case where the performance was half of the optimal. In the absense of infinite functional units, we showed a case where the performance was m/k of the optimal where m is the number of functional units and k is the number of cycles taken by the instruction to execute. Also, by using CDB (and not using instant writeback) we showed a case where the performance loss was proportional to the size of instruction code.

Later we proved Tomasulo's algorithm to be optimal under this set of conditions. This way we also proved the assumption set to be sufficient. This was done using the principle of mathematical induction.

An out-of-order issue would be a plugin to Tomasulo's algorithm, but we can improve if we use out-of-order dispatch. If we have infinite functional units, using in-order dispatch or out-of-order dispatch gives the same performance. But, in the practical case of finite number of functional units, having in-order dispatch can hamper performance. Also, we compare it against another algorithm which too has finite number of functional units, but uses out-of-order dispatch. Here we find a performance loss of $(2 - 1/m)$ by doing a worst case analysis where m is the number of functional

units. Thus, by using in-order issue but out-of-order dispatch, we can improve the performance of Tomasulo's algorithm over finite hardware.

References

[1] T. Arons and A. Pnueli. Verifying tomasulo's algorithm by refinement. *Proc. 12th International Conference on VLSI Design*, 1999.

[2] W. Damm and A. Pnueli. Verifying out-of-order executions. *D. Probst, editor, CHARME'97.*

[3] Silvia M. Muller Daniel Kroening and Wolfgang J. Paul. A rigorous correctness proof of a tomasulo scheduler supporting precise interrupts. *Proc. of the SCI'99/ISAS'99 International Conference.*

[4] Robert B. Jones Jens U. Skakkebaek and David L. Dill. Formal verification of out-of-order execution using incremental flushing. *Proc. 10th International Conference on Computer Aided Verification*, 1998.

[5] K. L. Mcmillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. *Proc. 10th International Conference on Computer Aided Verification*, 1998.

[6] Mandayam Srivas Ravi Hosabettu and Ganesh Gopalakrishnan. A proof of correctness of a processor implementing tomasulo's algorithm without a reorder buffer. *CHARME'99.*

[7] Mandayam Srivas Ravi Hosabettu and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. *Hu and Vardi*, 1998.

[8] Mandayam Srivas Ravi Hosabettu and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. *Proc. 11th International Conference on Computer Aided Verification*, 1999.

[9] J. Sawada and Jr. W. A. Hunt. Processor verification with precise exceptions and speculative execution. *Hu and Vardi*, 1998.

[10] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), 1967.

[11] R. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, Vol. 45, pp. 1563–1581, 1966.