Communicating Process Architectures – 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

Scheduling for ILP in the 'Processor-as-a-Network'

D. K. Arvind and S. Sotelo-Salazar
ICSA, School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland

Abstract. This paper explores the idea of the processor as an asynchronous network, called the micronet, of functional units which compute concurrently and communicate asynchronously. A micronet-based asynchronous processor exposes spatial as well as temporal concurrency. We analyse the performance of the 'processor-as-a-network' by comparing three scheduling algorithms for exploiting Instruction Level Parallelism (ILP). Schedulers for synchronous architectures have relied on deterministic instruction execution times. In contrast, ILP scheduling in micronet-based architectures is a challenge as it is less certain in advance when instructions start execution and when results become available. Performance results comparing the three schedulers are presented for SPEC95 benchmarks executing on a cycle-accurate model of the micronet architecture.

1 Introduction

The instruction execution times in a clocked or synchronous processor is fixed at the design stage. They are expressed in terms of numbers of clock cycles, which is both precise and deterministic (except, perhaps, in the case of memory instructions with caches), which forms the basis of compiler optimisations for Instruction Level Parallelism (ILP). In contrast, in a clock-free, micronet-based, asynchronous architecture [1], the operations proceed at their own speed which is dependent on the nature of the data, local delays and the availability of architectural resources. As a result the execution times of instructions would vary, albeit within a range, in a non-deterministic manner. This poses an interesting problem for an ILP compiler, which can no longer assume exact and deterministic instruction execution times for the purposes of scheduling and optimisation.

Instruction scheduling for architectures with different types of resources is known to be NP-hard, and a large body of work exists for clocked processors [2] [3] [4] [5]. This paper analyses the performance of the 'processor as a network', and, in particular, strategies for ILP scheduling in a micronet-based asynchronous processor (MAP). The PTD scheduler [6] had previously been proposed for scheduling instructions for architectures with uncertain latencies. The performance of the PTD scheduler is evaluated against two well-known list schedulers - the Gibbons-Muchnik (GM) [3] and the Balanced [4] schedulers. In the rest of this paper: Section 2 describes the architecture of the micronet-based processor; Section 3 presents scheduling strategies for ILP in micronet-based architectures, and the algorithmic complexity of the PTD scheduler is derived; Section 4 describes the Evaluation Framework which produced the results presented in Section 5.

2 Micronet-based Asynchronous Processor (MAP) Architectures

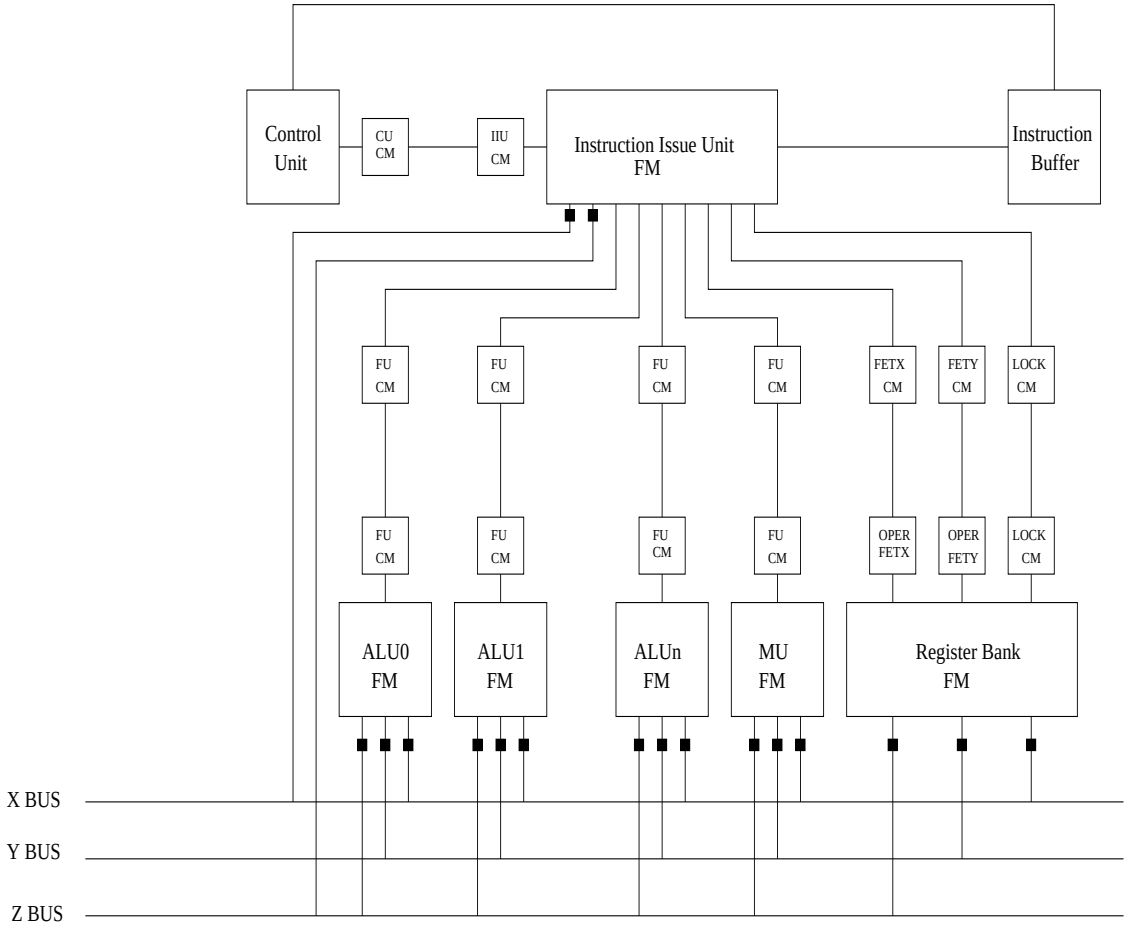


Figure 1: A MAP datapath

The micronet is a network of entities which compute concurrently and communicate asynchronously, with minimal centralised control. MAP is a family of processors based on the micronet operational model. The MAP datapath, as shown in Figure 1, is a *network* of execution units, such as the Instruction Issue Unit, Control Unit, Register Bank, and functional units such as the Arithmetic Unit(s) (AU), Logic Unit (LU), Floating Point Unit (FPU) and the Memory Unit (MU). The 32-bit datapath executes a subset of the MIPS instruction set.

Each execution unit is composed of a *Functional Microagent (FM)* which executes a specific micro-operation, and communicates with other units via *Communicating Microagents (CM)*, by employing a four-phase handshaking protocol. The Instruction Buffer caches instructions for the Instruction Issue Unit, and the Control Unit mediates their operations. On the issue of an instruction, the appropriate control signals are asserted and the destination register is locked, and the instruction is considered issued once the signals have been acknowledged. The CMs between the register bank and the X and Y buses communicate to place the register values, as specified in the instruction, on the X and Y buses, respectively. The ALU will operate on the values and the results are placed on the Z-bus and written back to the destination register, which is then unlocked. In its simplest form, the MAP architecture does not assume queues for the operands and the results in the execution units.

The issue unit issues one instruction at a time in an in-order manner. It is responsible for issuing instructions as soon as the operands become ready, *i.e.*, when they are written

Component Type		Latency time		Cycle time	
		Minimum	Maximum	Minimum	Maximum
Issue unit	(IU)	1.00	2.00	0.50	1.00
Read buses	(RF)	2.00	4.00	0.50	1.00
Write buses	(RF)	2.00	4.00	0.50	1.00
Arithmetic unit	(AU)	4.00	8.50	0.50	1.00
Logical unit	(LU)	2.00	7.00	0.50	1.00
Floating point unit	(FPU)	6.00	8.00	0.50	1.00
Memory unit	(MU)	10.00	20.00	0.50	1.00

Table 1: Latency and Cycles times (in ns) for the architectural components

into the register file. When the operands are ready, the issue unit inspects if both *read* buses and a functional unit of the appropriate type are all available. If this is the case, then the instruction is issued; otherwise, it is stalled, and the outcome depends on which resources were unavailable, as described next.

If a functional unit of the appropriate type is unavailable, but the *read* buses and operands are, then the instruction will be issued and the operand fetch will proceed normally. It will stall once the operands are read, and will remain so until the functional unit becomes available to execute the operation.

The second scenario occurs when any of the source operands are not ready (data requirement), or the *read* buses are unavailable (resource requirement). In either case, both the issue unit and the instruction will remain stalled until the data and resource requirements are satisfied.

Once a functional unit completes its execution, the result is sent to the register as soon as the Z bus becomes available. Otherwise, the functional unit stalls and remains in a "busy" state until it can deliver its result to the register.

A register *locking* scheme is employed to ensure that data coherency is maintained during the execution of the instruction. Such a scheme guarantees correctness of the data in the presence of asynchronous accesses to the register file. In MAP, each register has a lock bit which when set disables any read or write accesses to that particular register. Since there is no guarantee as to when a register will be unlocked, the issue unit cannot issue instructions which are dependent on a locked register, *i.e.*, if there is a RAW or WAW dependency, and will therefore remain stalled.

Conversely, when the issue unit proceeds to issue an instruction, its destination register is locked until the functional unit commits the result of the instruction to the register. If a subsequent instruction is stalled waiting on that data, then it will change status and issue if the other conditions are satisfied and will in turn lock the destination register of that instruction.

The delay values for the architectural components, as listed in Table 1, capture the range of latency values for the functional units, together with their cycle times, which is defined as the delay for components to become ready after completion of an operation. The distribution of latency times over the interval is determined by the type of functional unit, *e.g.*, the memory unit has a bimodal distribution which models the cache behaviour - the extreme values represent cache hits and misses; the distribution of latencies for the arithmetic units is based on the graph in Figure 4 in [7]; and the distribution is assumed to be uniform in the case of the Logic Unit.

2.1 Analysis of the MAP architecture

The MAP architecture, as shown in Figure 1, is a scalar architecture that features a single, in-order, instruction issue unit, with the results being committed fully out-of-order. Once instructions are issued, they proceed at their own speed, which is determined by the availability of data and resources. More than one instruction in flight implies that instructions may overtake.

The MAP architecture is a hybrid of certain features of VLIW and superscalar classes of architectures. Like VLIW architectures, the issue unit in the MAP architecture relies on the compiler to statically determine the schedule of independent instructions for execution. Similar to superscalar architectures, the MAP architecture is required to prevent run-time hazards. It differs from VLIW architectures in that it does not have to issue a number of independent instructions or no-operations in lock-step. Instead, the MAP architecture issues instructions as soon as the resources and data dependencies allow it to, and relies on the compiler to identify a schedule which minimises the instruction issue stalls and deals instructions at a rapid enough rate to keep all the functional units busy.

The compiler takes the strain in identifying a schedule which minimises the program execution time by maximising the instruction level parallelism and minimising stalls either in the issue unit or the functional units. Unlike a superscalar architecture, dyanamic data-forwarding is not implemented as this would require the operand fetch stage to be synchronised with the write-back stage. This synchronisation will inevitably slow down the faster stage, *i.e.* the fetch stage, when two dependent instructions are fetched and issued in succession. In such cases this synchronisation will take place whether or not data forwarding is implemented. However, if instructions are not scheduled one after the other, or if there is more than one instruction awaiting the result of the first instruction, then the write-back stage of the first instruction will be held up unnecessarily.

Once an instruction is issued in a micronet model, *i.e.* data and resource requirements are fulfilled, then it runs to completion without any synchronisation. Unlike VLIW architectures, there is no need to include "bubbles" in the schedule which is the case when no-ops are issued if the compiler cannot identify independent instructions for those time slots, *i.e.* those functional units will remain idle for an entire instruction cycle. Micronets have no need to issue no-ops which do not contribute to the execution of the program; instead, the compiler for the MAP architecture is required to schedule independent instructions such that the issue unit does not stall, or stalls minimally if it does.

The following section introduces the scheduling problem for a MAP architecture and the proposed solutions.

3 Instruction Scheduling for MAP Architectures

For a general, p-functional unit processor, the complexity of the scheduling problem using a non-preemptive approach is NP-complete (p is assumed to be greater than 2; in the case of $p = 2$, the complexity is polynomial, if the functional units are identical and the latencies are all the same). The problem is NP-hard in the case of a processor with functional units of different types and differing latencies. Scheduling hueristics are therefore employed which exploit domain knowledge to prune this exponentially increasing search space. The *list* scheduler uses a priority list of *ready* instructions. The hueristic will decide the best candidate instruction to be scheduled next, and when the instruction is removed then all its successors become available for consideration in the next cycle.

Gibbons and Muchnick [3] is an example of a list scheduler in which the ready instructions are prioritised on the basis that the candidate instruction will not cause an interlock with

Types of dependencies	Consecutive instructions	Separated by one inst.
True dependency with a load instruction	3	1
True dependency with a branch instruction	2	0
Any other true dependency	1	0
Resource dependency from a memory instruction	1	0

Table 2: Degree of the penalties depending of the type of dependency

.

the previous one, and given a choice, the candidate instruction is more likely to interlock with instructions after it.

The *Balanced* scheduler [4] was originally devised to take account of unpredictable memory access latencies. It is based on the idea of computing weights for load instructions based on the number of available independent instructions. These are scheduled, as in a traditional list scheduler, with independent instructions being distributed behind loads to buffer against unpredictable memory accesses. This idea is generalised to a MAP architecture in which all the instructions have unpredictable latencies. The priority for ready instructions is based on a weighted sum of values derived from heuristics tailored to the micronet model. These include the following: whether the instruction uses the same resources as the previous scheduled one; the number of immediate successors of the instruction; the length of the longest path from the instruction to the leaves of the graph; and, the number of source registers which are freed should the instruction be scheduled, which effectively takes account of register pressure.

The Penalise True Dependence (PTD) scheduler [6] had been proposed as a method for scheduling instructions for MAP architectures. The essence of the scheduler is to identify true data and resource dependencies in an instruction schedule and re-order the instructions where possible, so as to reduce the stalls in the instruction issue unit due to them. The PTD scheduler calculates a *penalty measure* which reflects the degree of resource contentions and stalls due to data dependencies. The scheduler moves instructions which would lower the penalty measure (Table 2 lists the degree of the penalties). The scheduler traverses the schedule to evaluate optimisations on every instruction that is penalised. In order to reduce the penalty measure the scheduler must find independent and unrelated instructions to place in between the dependent ones.

The instruction schedule may be visualised as a "horizontal" sequence of instructions which are executed in order, from left to right. When a penalised instruction is encountered, then an independent, unrelated instruction is searched on both sides of the offending instruction, first starting on the left, and if unsuccessful, switching to the right of the instruction. There are two necessary conditions for an instruction to be considered a suitable candidate for movement ahead of the penalised instruction. Firstly, the instruction in question has to be independent of the penalised instruction, and secondly, the instruction has to be independent of all instructions scheduled in between the candidate and the penalised instructions. These conditions are necessary to preserve the semantics of the code and are known as *valid* conditions. They allow only valid movement of instructions in which the order of execution is preserved, but the performance of the outcome of such movements has to be analysed further. The *safety* conditions guarantee that a valid movement is chosen which results in the penalty measure being reduced.

Benchmark	Total lines	GM. Time	Bal. Time	PTD Time	Percen.
`intmm`	151	0.0836	0.0839	0.0680	22.94 %
`livermore`	1915	3.7947	3.8388	2.7817	36.42 %
`fract`	5336	3.5940	3.5588	2.1920	62.35 %
`li`	16832	8.7857	8.1169	4.4349	83.02 %
`puzzle`	936	0.7781	0.6369	0.4711	35.19 %
`compress`	1236	0.9757	0.7300	0.5494	32.87 %
`go`	83838	65.2829	60.1321	44.0446	36.53 %
`m88k`	34089	22.0481	20.6159	12.6411	63.09 %

Table 3: Average compilation times (in seconds) of the benchmarks

The main reason for searching candidate instructions starting from the left-hand side of the penalty is because there is a greater likelihood of finding one faster. The exit of the basic block can be seen as a synchronisation point, and may therefore offer fewer options, and conversely, the entry of the basic block may offer a greater scope for suitable candidates. The scheduler aims to first reduce penalties with the higher delay-costs to the issue unit, *i.e.*,reduce the higher penalty values according to Table 2, which corresponds to data dependencies due to loads. The second pass attempts to reduce penalties due to branch instructions in the basic block, if it ends with such an instruction. And, in the final pass, the scheduler tries to reduce the remaining penalties with a value one.

The complexity of the PTD scheduler is derived to be $\theta(et^2 + n - e)$ (see Section 3.1 for details), where e is the number of penalties and n is the number of instructions in a basic block, and t is the distances (in terms of number of instructions) between the penalised and candidate instructions. The PTD scheduler compares favourably with the other two, as its complexity is governed by the number of penalties in a basic block (rather than the number of instructions), and which reduces as the algorithm progresses. In comparison, the number of iterations is constant in the case of the other two traditional techniques, and is proportional to the number of instructions in the basic block.

Table 3 gives a comparison of the average compilation times for the three schedulers. It gives the average of five compilation times (in seconds) of the scheduling sections in the three schedulers (this was obtained using the `gethrtime` function from the `time.h` standard library). The last column represents the percentage improvement of the PTD scheduler against the faster of the other two schedulers. The compilation times are on average 39% faster, with notable improvements of more than 60% for the `fract` and `m88k` benchmarks, and reaching a peak of 83% in the case of the `li` benchmark. The standard deviations revealed a narrow range of variations in the compilation times.

3.1 Algorithmic Complexity of the PTD Scheduler

The structure of the PTD scheduler is different from the GM and Balanced schedulers as the algorithm is driven by the penalties in the code. The complexity of the PTD scheduler is derived next.

The `while` sections in the functions *PTD_resource_phase* (Algorithm 1 in Appendix A), *PTD_consecutive_phase* (Algorithm 2) and *PTD_nonconsecutive_phase* (Algorithm 3) traverse the basic block stopping at every penalty. These sections have a complexity of $\theta\,(\,et^2\,+\,(n\,-\,e)\,)$, where n is the number of instructions in the basic block, and e is the

number of penalties. The term, et^2, corresponds to the time spent searching for a candidate instruction and checking its dependencies when the functions *PTD_arrange_left* and *PTD_arrange_right* are called. The term, $(n - e)$, covers the instructions that are not penalised.

The `repeat` loops (lines `3-16`, `17-30` and `31-44` in Algorithms 1, 2 and 3, respectively) ensure that at least one scheduling pass is performed. The `repeat` sections continue until there are no reductions in the penalty measure. The number of times these sections are repeated is denoted by c. Thus, the above term becomes $\theta\,(\,cet^2 + c\,(n - e)\,)$.

However, the parameters c, e and t are not general. Basic blocks have different number of penalties (e) of a given type and particular number of retries (c) for each basic block. Similarly, penalties have different instruction distances (t) when searching for a candidate.

The `repeat` block is replicated eight times in the three scheduling phases. Therefore, the complexity of the PTD scheduler is

$$\theta\,(\,8\,cet^2 + 8\,c\,(n - e) + 8\,nc + 3\,n\,) \tag{1}$$

The terms, $8\,nc$ and $3\,n$, represent the computation of the penalty measure throughout the three scheduling phases, both inside and outside the `repeat` sections, respectively.

Equation 1 has a few important simplifications. Observations of the scheduling process show that the number of times the `repeat` sections are looped is not greater than three or four. Therefore, c can be considered to be a small constant ($c = 2, 3, 4$). Since the search for candidate instructions start from the instruction neighbouring the penalised one, it is expected that the search (on either side) does not reach $n/2$. Furthermore, if the candidate is found on the left-hand side of the penalised instruction, the search on the right-hand side is not necessary. Thus, the factor t can be considered to be: $t \ll n$. As for the term e, it is often the case that there are not as many penalties of the same type as there are instructions, which results in $e < n$.

The upper and lower bounds of Equation 1 are defined by two opposite scenarios. The upper bound is represented by pure sequential code. It takes place when there are as many penalties as instructions ($e = n$), and there are no candidates found for all of those penalties ($t = n - 1$) and there can only be two retries ($c = 2$). The upper bound of Equation 1 is therefore

$$\theta\,(\,16\,n\,(n - 1)^2 + 19\,n\,) \tag{2}$$

However, since the same type of penalty cannot affect more than one `repeat` section, only one term is governed by $cn\,(n - 1)^2$, while the other seven share the term $c\,n$. The former term dominates, so the equation has an upper bound of n^3. In practice, the upper bound becomes:

$$\theta\,(\,2\,n\,(n - 1)^2 + 33\,n\,) \tag{3}$$

The lower bound of Equation 1 is represented by a purely independent code. There are no penalties in this case ($e = 0, \Rightarrow t = 0$) and no retries ($c = 0$). The complexity is therefore

$$\theta\,(\,3\,n\,) \tag{4}$$

The `if` conditions before the `repeat` sections avoid any scheduling attempt if there are no penalties. Only the initial PTD measures take part in the complexity. The lower bound of the PTD scheduler is therefore of the order of n.

If all the constants are removed from Equation 1, then the complexity of the PTD scheduler becomes

Benchmark	Functions	Basic Blocks	Instructions	Initialisation phase
intmm	6	11	196,074	12.000 %
livermore	16	48	1,601,672	84.504 %
fract	70	505	3,392,216	2.421 %
li	386	2,228	15,207,508	2.007 %
puzzle	20	137	16,055,217	0.509 %
compress	18	128	16,269,122	20.847 %
go	396	8,880	18,242,718	7.906 %
m88k	259	3,841	34,323,842	7.504 %

Table 4: Characteristics of the benchmarks

Benchmark	Arithmetic	Logic	Memory	Floating	Branch
intmm	70.11 %	5.13 %	20.04 %	0.00 %	4.72 %
livermore	73.75 %	6.60 %	13.35 %	0.00 %	6.30 %
fract	26.40 %	8.22 %	19.64 %	42.85 %	2.89 %
li	21.50 %	23.30 %	41.21 %	0.05 %	13.94 %
puzzle	52.08 %	22.12 %	11.43 %	0.00 %	14.37 %
compress	38.52 %	26.64 %	24.56 %	3.10 %	7.18 %
go	46.29 %	20.76 %	18.87 %	0.00 %	14.08 %
m88k	36.07 %	24.62 %	27.86 %	0.00 %	11.45 %
Average	45.59 %	17.17 %	22.12 %	5.75 %	9.37 %

Table 5: Breakdown of the benchmarks in terms of instruction types

$$\theta\,(\,et^2 + n - e\,) \tag{5}$$

In normal conditions, however, the parameters e and t have particular values with respect to n. As the algorithm progresses, the number of penalties is reduced, so e becomes: $e \ll n$. Similarly, t is much smaller than n ($t \ll n$), since, in general, the candidate is meant to be found from a close neighbour. Therefore, as the algorithm progresses in normal conditions, Equation 1 is found to be of the order of n.

4 The Evaluation Framework

The compilation environment is based on the Stanford University Intermediate Format (SUIF) compiler [8]. The benchmarks are characterised in Tables 4 and 5, and were derived mainly from the SPEC95 suite of benchmarks. The MAP architecture was modelled at the instruction level and simulated is an event-driven stochastic simulator that read and executed assembly language instructions generated by the SUIF compiler. Each instruction is associated with a number of events that emulate the stages in the micronet datapath for its execution. The events are created dynamically and their latency depend on the type of instruction and resource contentions at run-time.

The C benchmark programs are compiled in SUIF. Next, a loader program converts the resulting assembly code so that global memory references and labels fit into a global referencing scheme, and the resulting output is fed into the instruction-level simulator of the micronet architecture for evaluation. The output of this path is considered to be the *base* case since the code is not scheduled after code generation. Next the base case is fed into a

scheduling phase using one of the three schedulers and the output is fed to the simulator as before. Each result reported in the next section was the average of five runs.

5 Results

Figures 2 - 5 compare the reduction in the instruction issue stalls for the three schedulers for processor configurations consisting of one or more AUs. They represent the percentage improvement with respect to the base cases, *i.e.* unscheduled code executing on 1 AU, 2 AU, 3 AU and 4 AU, respectively. The causes of the issue stalls are broken down as due to general data dependencies (`Data`), data dependencies due to branch instruction (`Branch`), and those due to resource contention for the read buses (`Bus`), or for a functional unit (`Rsc`). The figures demonstrate that the three schedulers are successful in reducing the issue stalls due to data dependencies (`Data` and `Branch`) - the bars always lie along the positive y-axis. As the code is optimised, the causes of issue unit stalls shift from data dependencies to resource contentions (`Bus` and `Rsc`), which is to be expected as the functional units and their buses get busier. As the architecture scales the stalls due to bus contention become less important; in fact, these stalls are practically negligible in the 4 AU configuration (Figure 5).

The stalls due to contentions for functional units (`Rsc`) are also reduced as the architecture scales; however, their reduction is not as clear-cut as in the previous case as the scaling is confined only to arithmetic units. Benchmarks such as `li`, which have more memory instructions, cannot take advantage of the larger amount of concurrency in the AU units in architecture. The `fract` benchmark has a similar limitation since it contains a large proportion of floating point instructions (see Table 5). When the `fract` benchmark was simulated with four floating point units (FPU), the average reduction in the issue stalls due to resource contention went down from -17.91% (1 FPU) to -9.80% (4 FPU), in the case of the PTD scheduler.

The `livermore` benchmark is dominated by arithmetic instructions (see Table 5). The large negative percentages for this benchmark in the case of the 4 AU configuration (see Figure 5) presents scope for further improvement. This was confirmed for the PTD scheduler when the benchmark was simulated with a configuration containing 5 AU, and the issue stall was reduced from -11.22% (5 AU) to -10.18% (4 AU) (the remaining stalls are due to memory operations only). The `go` benchmark presents an interesting scaling pattern: for greater than 2 AU, the benchmark suffers a degradation in the `data` stalls which is mainly due to the limited parallelism in the benchmark. In contrast, the `bus` and `rsc` stalls improve as the architecture scales. `Puzzle` is recursive in nature which explains the high number of branch instructions in Table 5. In the absence of sufficient parallelism in the benchmark the issue stalls does not improve with an increase in the number of AUs. In contrast, the loop-oriented benchmarks such as `intmm` and `livermore` show significant improvement as the number of AUs are scaled.

A characteristic of the PTD scheduler is that it is more effective in reducing issue stalls due to resource dependencies than those due to data. This is clearly in evidence in Figure 3 with the `compress` benchmark. Both the GM and the Balanced schedulers have a significant impact on the issue stalls due to data dependencies (`data`), reducing it by almost 50%. The PTD scheduler does not exhibit the same increase (almost 40%), but it reduces the stalls due to resources to around -1%, whereas the other two schedulers manage a less respectable -10% reduction. In the case of the PTD scheduler, the improvements in the instruction issue stall due to data dependencies is restrained by the overlapping penalties. In contrast, the PTD scheduler tackles resource dependencies by applying penalties to consecutive instructions of the same type when there are not enough functional units of that type. The net result is that the overall improvement in the issue stall due to PTD compares well with the other two

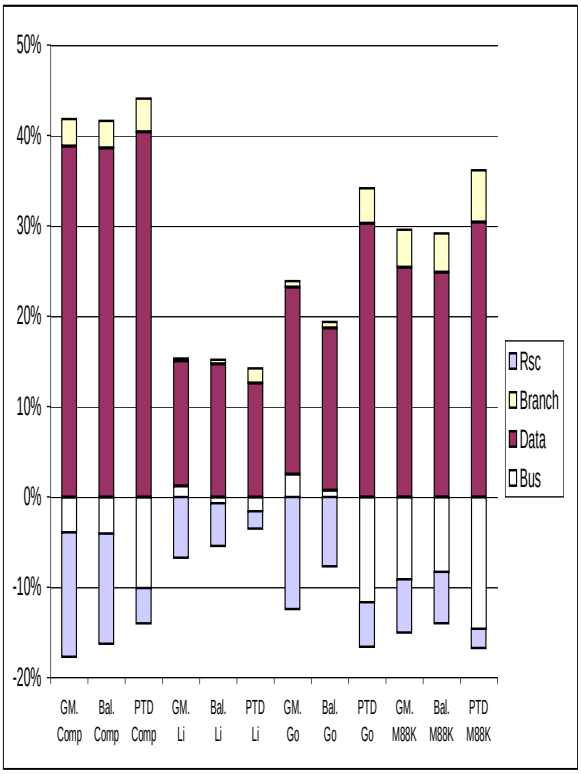


Figure 2: Percentage improvement in the instruction issue stalls (1 AU)

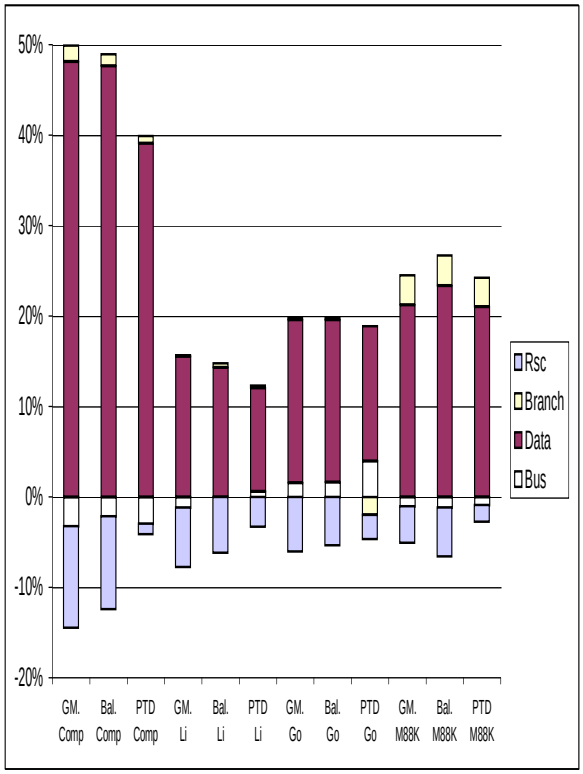
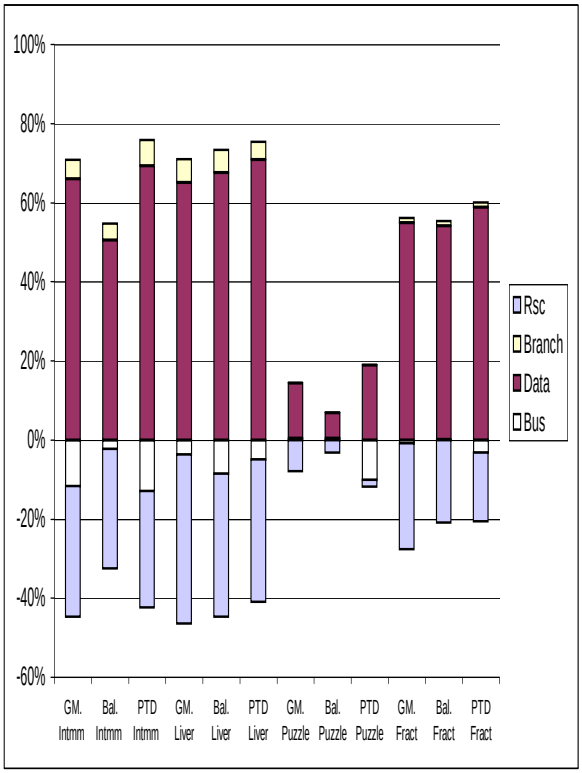
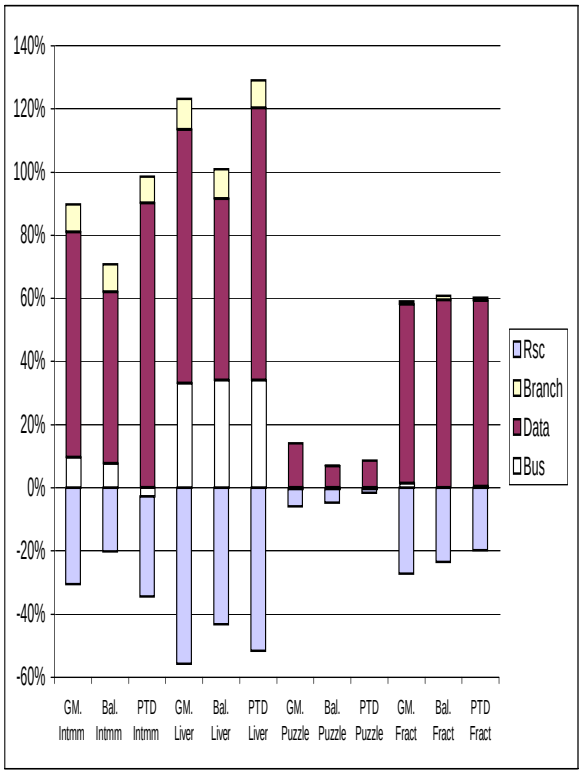


Figure 3: Percentage improvement in the instruction issue stalls (2 AU)





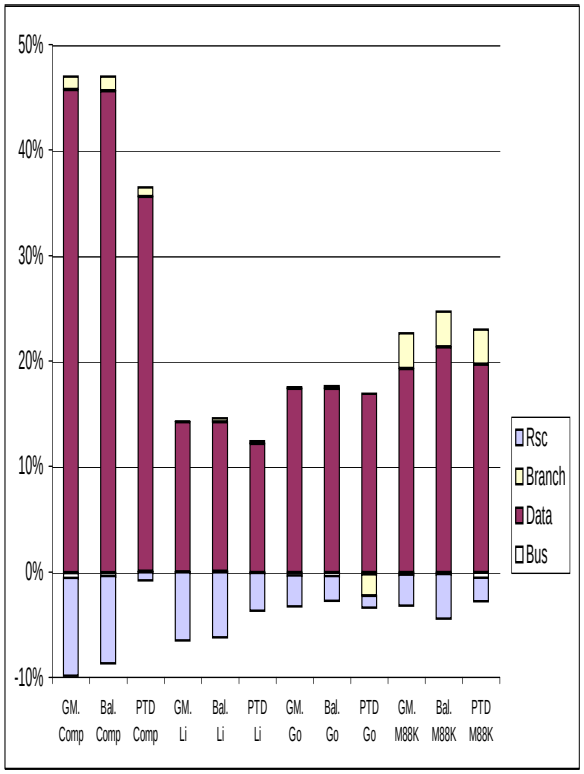


Figure 4: Percentage improvement in the instruction issue stalls (3 AU)

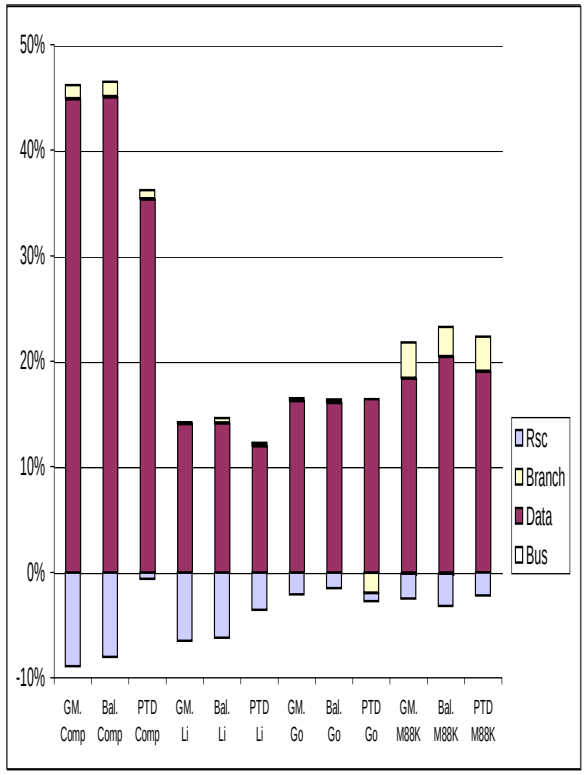
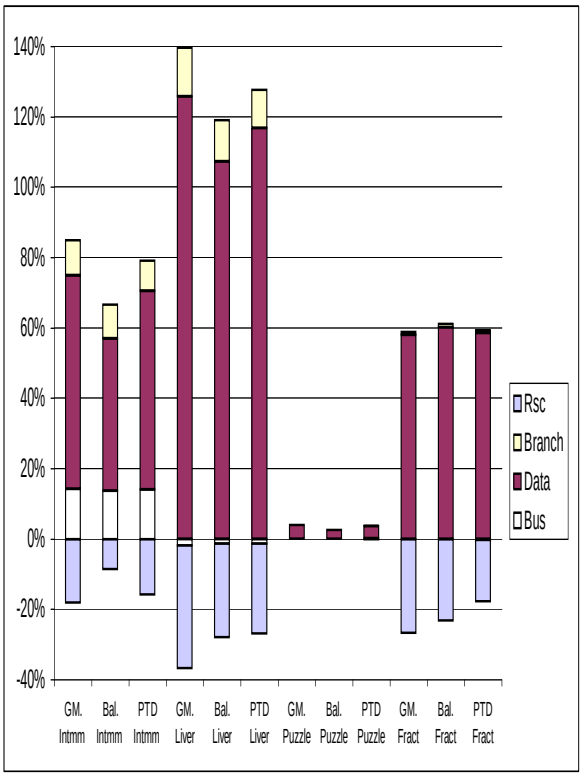
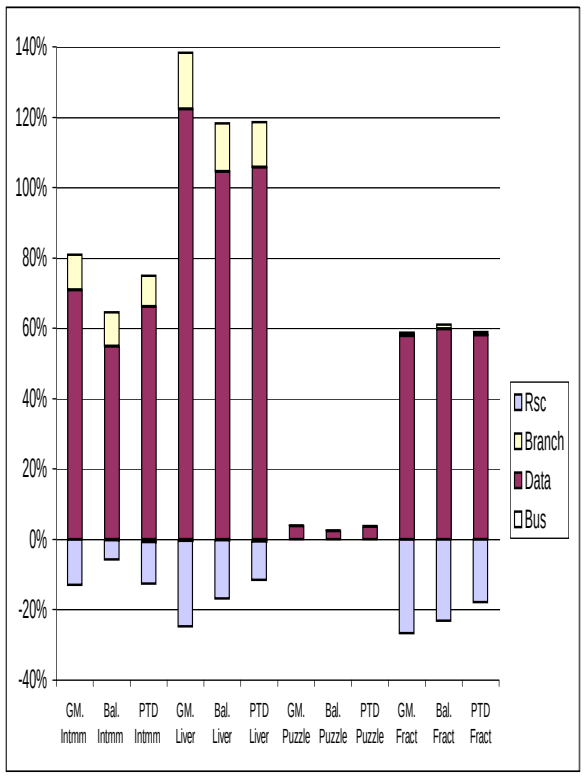


Figure 5: Percentage improvement in the instruction issue stalls (4 AU)





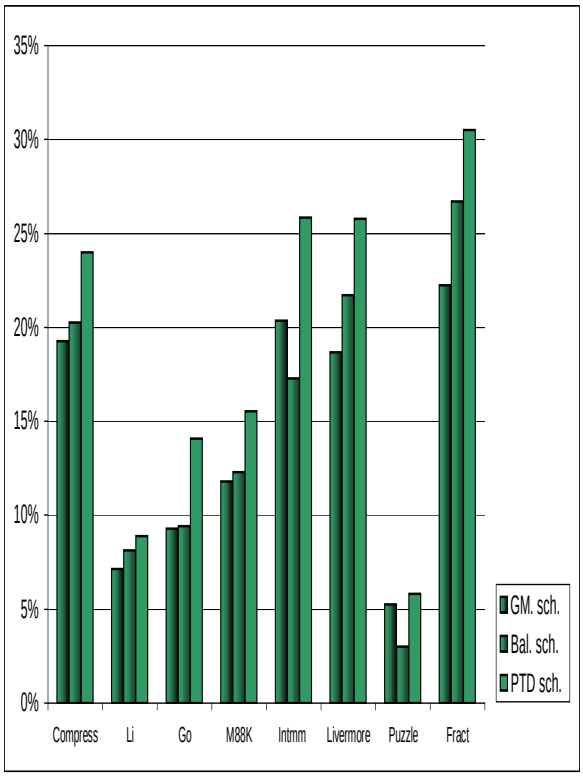


Figure 6: Percentage improvement in execution times due to scheduling for ILP (1 AU)

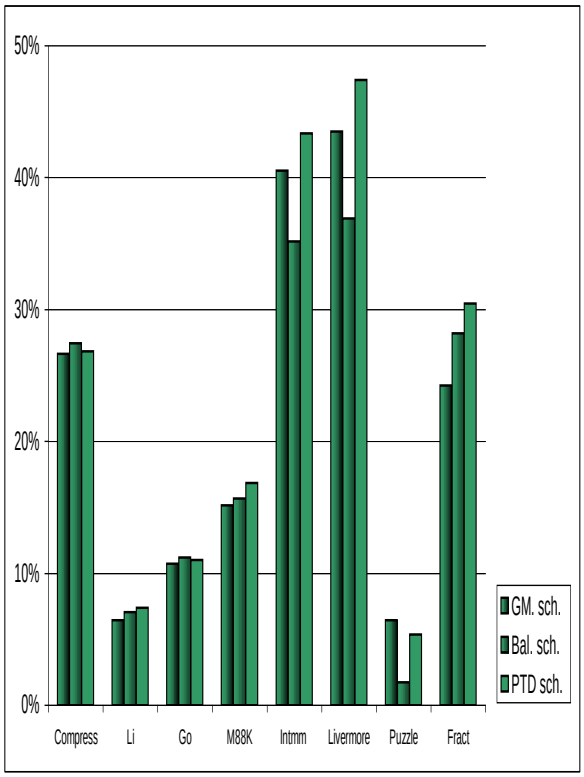


Figure 7: Percentage improvement in execution times due to scheduling for ILP (2 AU)

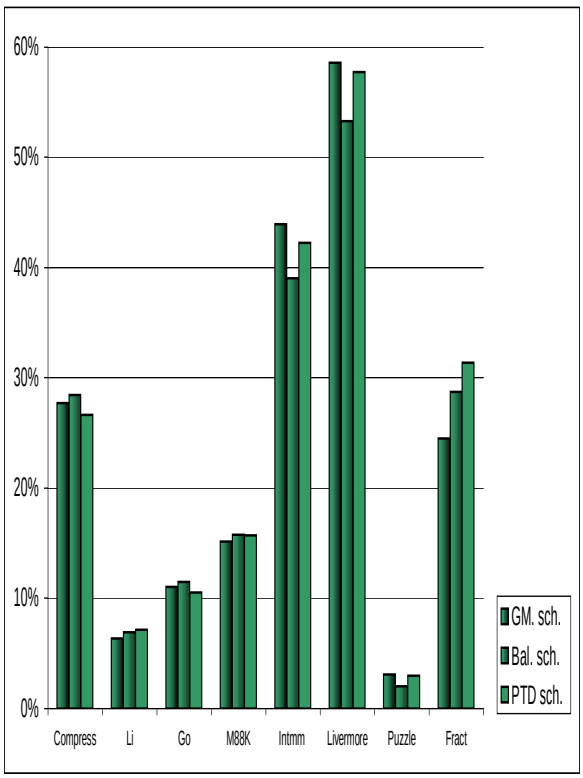


Figure 8: Percentage improvement in execution times due to scheduling for ILP (3 AU)

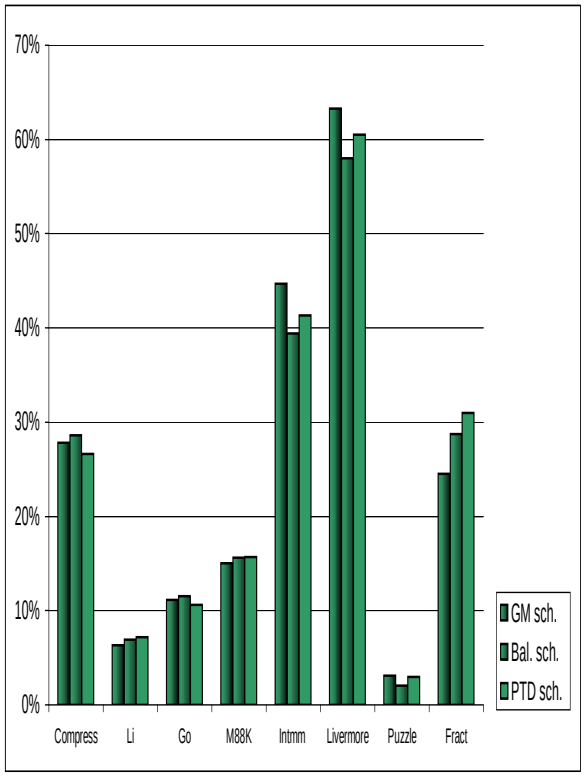


Figure 9: Percentage improvement in execution times due to scheduling for ILP (4 AU)

schedulers.

Figures 6 - 9 illustrate the percentage improvement in the program execution times due to instruction level parallelism for configurations ranging from 1 to 4 AUs. In each configuration, the improvement is compared against the unscheduled code executing on the same number of AUs. All the three schedulers benefited from memory disambiguation. The PTD scheduler consistently outperforms the other two by 4% on average in the 1 AU configuration. This falls to within 2% as the architecture scales. This effect is due to the overlapping penalties produced by the PTD scheduler as it optimises the code. They confirm the correlation between the reduction in issue unit stalls and the improvement in overall execution times, as predicted.

6 Conclusions

This paper has analysed the performance of three ILP schedulers for the 'processor-as-a-network'. Instruction scheduling for such an architecture is important in order to exploit fine-grain temporal and spatial concurrency. The PTD scheduler has addressed the problem of statically generating efficient instruction schedules for a micronet target with uncertain instruction latencies. The complexity of the PTD algorithm improves on other well-known list-based schedulers and performs better than them when resources are constrained and at least as well when the resources are scaled. In this paper the application of the PTD scheduler has been confined to a basic block. It has been demonstrated recently that PTD can be extended as a global scheduler in conjunction with techniques such as code motion beyond basic block boundaries to produce better schedules for micronet-based asynchronous architectures. This is both a feasible and attractive proposition thanks to the efficient compilation times of the PTD scheduler.

Acknowledgements

Salvador Sotelo-Salazar was supported by a postgraduate studentship from the Science and Technology National Council of Mexico (CONACYT).

References

[1] Arvind, D. K. and Rebello, V. 1994. Instruction Level Parallelism in Asynchronous Processor Architectures. In *Proc. of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pp 80-91, Eds. M, Moonen and F. Cathoor, Leuven, Belgium, Aug. 1994, Elsevier.

[2] V. Bala and N. Rubin. Efficient scheduling using finite state automota. In *28th Annual International Symposium on Microarchitecture (MICRO-28)*, pp 46-56, Nov. 1995.

[3] P. B. Gibbons and S. S. Muchnik. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN '86 Symposium on Compiler Construction*, 21(7):11-16, July 1986.

[4] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *ACM SIGPLAN'93, Language on Programming Language Design and Implementation*, pp 278-89, June 1993.

[5] K. V. Palem and B. B. Simons. Scheduling time-critical instructions on RISC machines. In *ACM Transactions on Programming Languages and Systems*, 15(4):632-58, Sep. 1993.

[6] D. K. Arvind and S. Sotelo-Salazar. Scheduling instructions with uncertain latencies in asynchronous architectures. In *3rd International Euro-Par Conference*, pp 771-78, Springer-Verlag, Passau, Germany, Aug. 1997.

[7] D. J. Kinniment. An evaluation of asynchronous addition. In *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 4(1):137-40, March 1996.

[8] M. W. Hall *et al*. Maximizing multiprocessor performance with the SUIF compiler. In *IEEE Computer*, Dec. 1996.

Appendix A

Algorithm 1: PTD_resource_phase_root Algorithm

```
 1: measure = PTD_measure(root, resource_phase)
 2: IF measure > 0 THEN
 3:      REPEAT
 4:        node = root
 5:        last_measure = measure
 6:        WHILE node ~= NULL DO
 7:          IF penalty_resource(node) = 3 THEN {PENALTY: MEMORY INST.}
 8:            PTD_arrange_left_resource(node)
 9:          ENDIF
10:          IF penalty_resource(node) = 3 THEN {PENALTY: MEMORY INST.}
11:            PTD_arrange_right_resource(node)
12:          ENDIF
13:          node = next(node)
14:        ENDWHILE
15:        measure = PTD_measure(root, resource_phase)
16:      UNTIL measure = last_measure

17:      REPEAT
18:        node = root
19:        last_measure = measure
20:        WHILE node ~= NULL DO
21:          IF penalty_resource(node) = 1 THEN {PENALTY: OTHER TYPES}
22:             PTD_arrange_left_resource(node)
23:          ENDIF
24:          IF penalty_resource(node) = 1 THEN {PENALTY: OTHER TYPES}
25:             PTD_arrange_right_resource(node)
26:          ENDIF
27:          node = next(node)
28:        ENDWHILE
29:        measure = PTD_measure(root, resource_phase)}
30:      UNTIL measure = last_measure
31: ENDIF
```

Algorithm 2: PTD_consecutive_phase(root) Algorithm

```
 1: measure = PTD_measure(root, first_phase)
 2: IF measure > 0
 3:    REPEAT
 4:       node = root
 5:       last_measure = measure
 6:       WHILE node ~= NULL DO
 7:         IF penalty_consecutive(node) = 3 THEN {PENALTY: LOAD INST.}
 8:            PTD_arrange_left_data(node)
 9:         ENDIF
10:         IF penalty_consecutive(node) = 3 THEN {PENALTY: LOAD INST.}
11:             PTD_arrange_right_data(node)
12:         ENDIF
```

```
13:         node = next(node)
14:       ENDWHILE
15:       measure = PTD_measure(root, first_phase)
16:    UNTIL measure = last_measure

17:    REPEAT
18:        node = root
19:        last_measure = measure
20:        WHILE node ~= NULL DO
21:          IF penalty_consecutive(node) = 2 THEN {PENALTY: BRANCH INST.}
22:             PTD_arrange_left_data(node)
23:          ENDIF
24:          IF penalty_consecutive(node) = 2 THEN {PENALTY: BRANCH INST.}
25:             PTD_arrange_right_data(node)
26:          ENDIF
27:          node = next(node)
28:        ENDWHILE
29:        measure = PTD_measure(root, first_phase)
30:    UNTIL measure = last_measure

31:    REPEAT
32:        node = root
33:        last_measure = measure

34:        WHILE node ~= NULL DO
35:          IF penalty_consecutive(node) = 1 THEN {PENALTY: OTHER INST.}
36:             PTD_arrange_left_data(node)
37:          ENDIF
38:          IF penalty_consecutive(node) = 1 THEN {PENALTY: OTHER INST.}
39:             PTD_arrange_right_data(node)
40:          ENDIF
41:          node = next(node)
42:        ENDWHILE
43:        measure = PTD_measure(root, first_phase)
44:    UNTIL measure = last_measure
45: ENDIF
```

Algorithm 3: PTD_nonconsecutive_phase(root) Algorithm

```
 1: measure = PTD_measure(root, second_phase)
 2: IF measure > 0 THEN
 3:  REPEAT
 4:     node = root
 5:     last_measure = measure

 6:     WHILE node ~= NULL DO
 7:      IF penalty_nonconsecutive(node) = 3 THEN {DISTANCE: 1 INST.}
 8:         PTD_arrange_left_data(node)
 9:      ENDIF
10:      IF penalty_nonconsecutive(node) = 3 THEN {DISTANCE: 1 INST.}
11:         PTD_arrange_right_data(node)
12:      ENDIF
13:      node = next(node)
14:     ENDWHILE
15:    measure = PTD_measure(root, second_phase)
16:  UNTIL measure = last_measure

17:  REPEAT
```

```
18:      node = root
19:      last_measure = measure

20:      WHILE node ~= NULL DO
21:        IF penalty_nonconsecutive(node) = 2 THEN {DISTANCE: 2 INST.}
22:           PTD_arrange_left_data(node)
23:        ENDIF
24:        IF penalty_nonconsecutive(node) = 2 THEN {DISTANCE: 2 INST.}
25:           PTD_arrange_right_data(node)
26:        ENDIF
27:        node = next(node)
28:      ENDWHILE
29:      measure = PTD_measure(root, second_phase)
30:   UNTIL measure = last_measure

31:   REPEAT
32:      node = root
33:      last_measure = measure

34:      WHILE node ~= NULL DO
35:        IF penalty_nonconsecutive(node) = 1 THEN {DISTANCE: 3 INST.}
36:           PTD_arrange_left_data(node)
37:        ENDIF
38:        IF penalty_nonconsecutive(node) = 1 THEN {DISTANCE: 3 INST.}
39:           PTD_arrange_right_data(node)
40:        ENDIF
41:        node = next(node)
42:      ENDWHILE
43:      measure = PTD_measure(root, second_phase)
44:   UNTIL measure = last_measure
45: ENDIF
```