Static Scheduling of Instructions on Micronet-based Asynchronous Processors

D. K. Arvind and V. E. F. Rebello
Department of Computer Science, The University of Edinburgh
Edinburgh, EH9 3JZ, United Kingdom
E-mail: {dka, vefr}@dcs.ed.ac.uk

Abstract

This paper investigates issues which impinge on the design of static instruction schedulers for micronet-based asynchronous processor (MAP) architectures. The micronet model exposes both temporal and spatial concurrency within a processor. A list scheduling algorithm is described which has been optimised with MAP-specific heuristics. Their performance on some program graphs are presented and conclusions are drawn on the suitability of MAP as targets for ILP compilers.

Keywords: Asynchronous Processor Architecture, Instruction-level Parallelism (ILP), Micronets, Static scheduling.

1 Introduction

A number of novel asynchronous processor architectures have been proposed recently [6, 9, 10, 12, 21, 23, 24, 27], but scant attention has been paid to any understanding of the interactions between the processor and compiler designs. Instead, existing synchronous RISC compiler technology has been reused (largely unmodified), while exploiting any improvements in the performance of the hardware which asynchrony provides.

One of the outcomes of the RISC design approach had been a deeper understanding of the interactions between the processor design and the implementation and compiler technologies, respectively. The processors were streamlined for efficient implementation in the emerging VLSI technology, and the system complexity was migrated upwards to their compilers. For instance, MIPS did away with hardware interlocks and relied instead on the compiler to reorder instructions and introduce null ones where appropriate [15]. The optimisers for synchronous pipelines have assumed a deterministic model of the target, with each stage delay being approximated to being the same, having been fixed *a priori* by the clock. They produce, both, an order of execution for the instructions, and the times - in terms of multiples of the basic RISC instruction cycle, when they are to execute. In contrast, a linear, asynchronous pipeline, e.g. *micropipeline* [28], has stages whose delays can vary, thanks to data dependencies. Now, the compiler has a less accurate timing model of the target, and any optimisations based on a synchronous model, such as scheduling instructions in execution gaps, are less effective.

A *micronet* is a network of pipelines, with (selected) stages of different pipelines being able to communicate with each other. This enables the exploitation of both spatial and temporal concurrency between instructions [2] (in contrast, a micropipeline only exploits temporal parallelism [4]). It is more difficult for a compiler to predict the behaviour of the micronet for the following reasons: firstly, as in a micropipeline the delay of each pipeline stage might vary; secondly and more uniquely, each instruction only visits the relevant stages and the multiple paths enable more than one instruction to operate concurrently within a stage, which enables instructions to race each other, with possible out-of-order completion of instructions. Furthermore, instructions may interfere with each other when competing for the same resource in a particular stage.

The effective performance which a MAP system can deliver depends intimately on the compiler's ability to match the parallelism in programs with the temporal and spatial concurrency exposed by the MAP architecture. This paper is a preliminary attempt to understand the interface between the back-end of a parallelising compiler and MAP architectures. In the rest of this paper, Section 2 briefly describes MAP architectures; Section 3 introduces the MAP schedul-

Published in the Proceedings of the 2nd International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'96), pp 80-91, Aizu Wakamatsu City, Japan, March 1996.

ing problem, and presents MAP-specific optimisations to a list scheduling algorithm; Section 4 evaluates the quality of the resulting schedules; Section 5 discusses the choice of instruction costs and the effect of instruction interference on the scheduler; and finally, Section 6 provides concluding remarks and scope for future work.

2 A brief introduction to MAP architectures

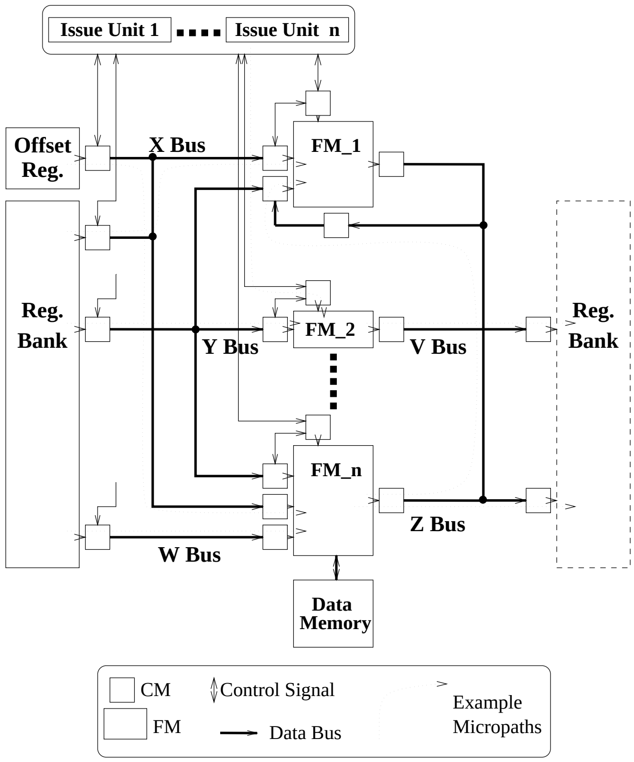

Figure 1: A micronet model of a MAP architecture

A *micronet* is an ensemble of *micropaths*, where a micropath is a pipeline or sequence of *microagents*, and in turn, a microagent performs either a *communicating* or a *functional micro-operation*. A functional microagent (FM) communicates with other FMs through their respective communicating microagents (CM). Even with a single issue unit, where the issue rate is faster than the slowest instruction execution rate, the microagents can all operate concurrently in space, in addition to the temporal concurrency associated with pipelines. Another feature of the micronet is that the micro-operations for an instruction are initiated independently by the issue unit, as soon as their particular microagents become available, and delegates all control to them, thus freeing the issue unit for the next instruction. Therefore, the idle time between instructions is kept to a minimum [2]. The executing instructions also release their microagents individually, as soon as the respective micro-operations have completed, thus freeing the resources immediately for another instruction. Finally, through a novel application of the communication protocol, datapath hazards are resolved efficiently while hiding the overheads of asynchrony [3].

A micronet can be stalled due to contention for resources. In particular, the issue unit will be stalled when the resources required by the current instruction are all busy. The scheduler attempts to minimise this by suitably ordering the instructions at compile-time. If it is impossible to schedule successive unrelated instructions, then the micronet minimises the stall at run-time. In the case of data-dependent instructions: both instructions are issued, with the second instruction awaiting the result to be forwarded. In the case of resource contention: the second instruction performs all the micro-operations up to the microagent which is busy. In effect, only the offending micro-operation is stalled, rather than the entire instruction. A detailed explanation of hazard avoidance is given in [3], with implications for the scheduler being detailed below:

Read-after-Write - Although the dependent instruction will be issued, its execution will be delayed until the completion of its predecessor. In practice, it is preferable not to issue such an instruction, since the resources earmarked for the dependent instruction are unavailable for use by other, now "ready-to-execute", instructions, which might introduce further structural hazards in the bargain.

Write-after-Write - The write-back order has to be maintained and this is achieved in hardware by the micronet. The two instructions are permitted to execute concurrently. Although all of the second instruction's microoperations will have been initiated, the write-back microoperation will stall for as long as the first instruction holds on to the destination register. The current MAP architecture supports only one outstanding register lock request, therefore a subsequent third instruction which requires a locked register cannot be issued, until the first write-back has been completed. The scheduler should avoid arranging instructions which write to the register file immediately after two instructions with write-after-write dependencies.

Write-after-Read - In the case of an architecture with a single set of operand fetch buses, the hardware ensures that a dependent instruction will be unable to lock its destination register before its predecessor has fetched its operand. Should there be a number of operand fetch buses (as in a superscalar MAP), and the possibility of a dependent instruction obtaining its operands before its predecessor, then this instruction may have to be stalled. This would only be necessary when the time to execute the dependent instruction is less than the operand fetch time for the predecessor. This hazard is also known as an anti-dependency, and along with write-after-write hazards can be avoided by register renaming.

Hazard resolution is a good example of the interaction between the compiler and the architecture. Since there is no concept of time in the schedule, it is impossible to avoid all hazards at compile time (c.f. the MIPS organiser). The scheduler can only hope to produce an ordering of instructions which reduces the number of hazards, and relies on the MAP architecture to minimise their effects by efficiently resolving them in hardware.

The computational model for synchronous RISC architectures is simple, in the sense that the execution times of instructions are considered fixed and instructions do not contend for resources. Neither of these hold for MAP architectures. The MAP model describes the architecture as a collection of microagents, where each one has an micro-operation latency which determines when the result of that micro-operation becomes available; and a micro-operation cycle time, which signifies the rate at which the micro-operations can be executed.

3 The MAP Scheduler

The MAP scheduling problem can be stated as follows: *Given a set of heterogeneous resources with variable execution times, devise a minimal-length, non-preemptive schedule which respects dependencies within programs. Each program being described as an arbitrary partial ordering of instructions.*

The precedence- and resource-constrained instruction scheduling problem has been studied well, and it is known that even by imposing restrictions, the problem is still NP-hard [7] [17] [29]. For example, when the execution times of tasks are not uniform and their partial order is arbitrary, then for two or more identical processing units, the problem of determining a minimal-length, non-preemptive schedule is NP-complete [13]. This result is true even if all of the tasks are independent. Therefore, in order to achieve near-optimal execution times for given applications on MAP architectures, an efficient (polynomial-time) scheduling algorithm based on one or a number of heuristics must be devised.

3.1 The MAP Scheduling Problem

List scheduling (LS) is a general method for scheduling tasks in resource- constrained problems [7]. LS builds a *ready set* that contains all of the tasks which are not waiting on the results of other tasks. When a processor becomes available, a task with the highest priority is chosen from the set and assigned to it. The ready set is obtained from a topological sort of the data dependence graph. LS relies on other heuristics to prioritise the ready tasks and guide it towards an optimal solution. This has lead to a profusion of LS-based heuristics [5, 11, 16, 20, 25].

The MAP solution adopted here is based on the optimal, greedy scheduling algorithm for list scheduling which was proposed by Coffman and Graham [8]. This is an $O(n^2)$ algorithm for arbitrary precedence constraints for two processors with unit execution costs. A MAP scheduler has to deal with heterogeneous resources and can no longer just choose the ready instruction with the highest priority, but must also consider whether the correct resources are also available. Once an instruction is issued, its execution cannot be suspended and resumed at the point of suspension at a later time, i.e. schedules must be non-preemptive. The goodness of these schedules are highly dependent on the parameter(s) that are used to prioritise instructions within the ready list [1] [22], and these are next discussed.

3.1.1 Minimising Idle Times

The scheduler's first assumption is that minimising the stall time will lead to an optimal (or at least near-optimal) program execution time (the first priority heuristic). This implies that the MAP compiler should not schedule instructions until the required microagents (resources) are available. Also, the hazards due to data dependencies outlined in the previous section should be avoided. All of this implies that the computational model has to maintain a scoreboard of resource activities.

3.1.2 Primary Instruction Priority

In Coffman and Graham's algorithm, interprocessor communication is assumed to be zero and tasks have unit execution times, which means that time can be conveniently treated as being discrete rather than continuous. This allows priorities to be assigned based on the task's level within the DAG from the sink tasks. Since instructions have different worst-case execution times in MAP, the problem is similar to multiprocessor scheduling with interprocessor communication delays (where communication costs are only incurred if dependent tasks are scheduled on different processors). The solutions adopted in this field have been based on critical path analysis and heuristics [14] [19] [26]. (The critical path cost of a task is the largest sum of costs along a path from itself to a sink task.) In the MAP computational model, although actual instruction execution costs may vary, these critical path costs can be determined *a priori* by basing them on fixed, worst-case instruction costs.

3.1.3 Secondary Instruction Priority

The heuristics applied so far may still not prioritise the executable tasks (i.e. those tasks whose operands and resources are available and are therefore ready for execution) sufficiently. One feature which does seem to significantly influence the best choice of candidate is the dependents of the chosen task. The two heuristics used to "break ties" amongst candidates of the same priority act as follows: the first one gives a higher priority to the task with the larger number of successors which are *solely* dependent on it. If a tie is still unbroken, then a higher priority is given to the task with the most number of successors. A feature of these heuristics is that the priority of a task increases with time. Additionally, these heuristics highlight the need to consider not only which tasks need to execute in the future, but also their resources.

3.1.4 Importance of the Instruction Issue Cycle Time

Unlike synchronous pipelines, micronet resources have two parameters which affect instruction execution costs: the microoperation's latency and its cycle time [4]. Latency determines when data becomes available for subsequent micro-operations and other instructions. The cycle time influences when a resource (microagent) becomes available again for use by a subsequent instruction (or microoperation). Together with program parallelism and the number of resources, a limiting factor on the amount of exploitable ILP is the cycle time of the issue unit in relationship to the execution time of instructions (or more accurately their cycle times).

In order to minimise the issue unit's stall time, the compiler has to devise a schedule that allows instructions to be issued continuously at the highest possible rate, which is equivalent to one every minimum Instruction Issue Cycle Time (IICT) [3]. Traditional synchronous datapaths are pipelined or where necessary super-pipelined (i.e. the functional units are themselves pipelined) sufficiently to achieve this goal. Due to the spatial ILP in MAP, instructions are issued at a rate (determined by the IICT and dependencies) which is faster than their Instruction Cycle Times (ICTs). The ICT is the effective issue time (due to pipelining) for a particular instruction, which is determined by the rate at which that specific instruction type can be processed. As the IICT, which is less than the largest ICT, gets smaller, the MAP architecture behaves more in a superscalar fashion and therefore the value of the IICT itself can have a significant influence on the optimality of a schedule. This is less significant when the IICT is comparable to the largest ICT, in which case the order of the independent instructions is less critical, since the micronet behaves like a linear pipeline without any spatial concurrency.

3.1.5 IICT, ICT and Lookahead

When choosing an instruction to schedule, it may be beneficial to consider not only those instructions which are ready, but also ones which will become ready in the near future, called *instruction lookahead*, e.g. within the next minimum IICT. Note that this may mean deliberately selecting an instruction that causes the processor's issue unit to stall.

Another form of lookahead is to consider the future resource requirements when scheduling instructions, called *resource lookahead*. The two steps in choosing an instruction and checking for availability of resources should take place in conjunction (See Algorithm 1 for more details).

3.2 The MAP scheduling algorithm

The algorithm takes as its input a directed graph of instruction dependencies and a resource graph with architectural parameters, and generates an instruction schedule for the given MAP architecture. Two lists are defined as follows: the WI_list - the list of instructions still awaiting their operands, and the EI_list - an ordered list of instructions which are ready, or will be ready in the near future (for lookahead instructions), but still awaiting issue. The order of the latter list is determined by the critical path costs of instructions, i.e. the primary priority. Next, a prioritised list of executable instructions is derived from the EI_list based on the availability of their resources at the current time. If there are ties, an instruction (or instructions in the case of superscalar MAP) is chosen for issue based on secondary priority values.

The scheduler mimics the behaviour of the architecture's issue unit. The function *generate_schedule()*, as shown in Algorithm 1, schedules instructions based on their readiness, their priority and the availability of resources. Unlike schedulers for synchronous machines, the scheduling of instructions does not proceed in uniform time steps, but rather in an asynchronous event-driven manner until all the instructions are scheduled. Each iteration of the main loop (the while do loop in line 5) corresponds to an instant in time when the issue unit is ready to issue an instruction. However, a situation may arise when at some given time there are no instructions ready for issue (line 8), in which case the clock must be advanced, but only as far as necessary to remedy this. The incrementing of the clock simulates the issue unit being stalled. The routine, *advance_clock()*, finds the earliest occurrence of three types of events: the ready time of an instruction in the WI_list and of a lookahead instruction in the EI_list; the time when the result of an operation becomes available in the register file; and the time a busy resource becomes free. Only the first two events can change the status of the EI_list. There is a choice of heuristics which can be applied, either the instruction lookahead or the traditional priority-based approach. Instruction lookahead (lines 9 - 17) chooses the best instruction to issue from the EI_list based on the lookahead heuristic. The function, *get_ready_instr()*, returns from the given list of instructions the one with the highest estimated-time-to-completion (ETC) priority for which there will be sufficient resources in the datapath if it is issued at its earliest issue time. This time may be the current issue time or some time in the future. In the case of the latter, issuing this instruction will cause the issue unit to stall. In the current implementation of the lookahead heuristic, only one instruction is chosen per issue cycle iteration. The routine, *apply.lookahead()*, implements the instruction lookahead heuristic which uses the ETC priority and the earliest issue time of two instructions to determine which of them should be issued first. By comparing the estimated execution time of the two instruction schedules, the order with the smallest time is chosen. Should the two schedules have the same time, then the order where an instruction completes the earliest is chosen, since this allows dependents to become ready sooner. The alternative heuristic (lines 18 - 29) chooses the instruction with the highest priority which can be issued immediately. This may involve choosing one or more from a number of instructions with the same primary priority value (ETC). Line 19 creates a list of ready instructions with the same, highest ETC values and line 22 removes those instructions with insufficient resources for issue at the current time. Line 23 supports architectures which incorporate lockstep superscalar instruction issue. The routine *issue_all()* issues as many of the instructions as possible from the given list. If there are not enough issue-slots for the complete list (*readyI*), then the routine *choosing_insts()* returns the best instruction for issue based on the secondary priorities. The two loops (lines 26 and 27) repeat until either the issues slots are filled or their respective lists become empty. The clock is advanced appropriately depending on whether or not the scheduler was able to issue one or more instructions at the current time (lines 28 and 29). The routine, *update_writeback*, models the behaviour of the portion of the micronet not directly controlled by the issue unit, e.g. write-back bus. Line 32 updates the instruction lists and the next instruction issue cycle iteration begins at a new time.

Example 1 and Example 2 contrast the influence of IICT and resource lookahead on determining an optimal schedule. A_1 and B are ready candidate instructions, with a third instruction, A_2, which has a structural dependency on A_1.

The lookahead heuristics attempt to match the available program and architectural parallelism over a short window of time. The strategy of repeating the process over the entire program allows the instruction-level parallelism to be exploited more evenly. This has two effects: firstly, a better program makespan is usually achieved; secondly, a schedule is generated which is more robust to deviations from the predicted instruction costs because only the appropriate amount of program parallelism is exposed which can be exploited by the target at any one time. Since costs are based on worse-case values rather than typical ones, the traditional list scheduling heuristics tend to overly migrate independent instructions to the top of the schedule, leaving insufficient parallelism for later. Kerns and Eggers [18] proposed a code scheduling algorithm called *balanced scheduling* for synchronous architectures which is similar in concept. Their algorithm is specifically designed to tolerate a wide range of variance in load latency, e.g. cache misses/hits, global and

Algorithm 1 : ***The MAP scheduler (generate_schedule())***

```
curr_time := 0;
calc_completion_times(); \* Critical path analysis for each instruction *\
update_WI(WI_list); \* Determine instruction start times *\
update_EI(WI_list); \* Move ready instructions to EI_list *\
while (WI_list ≠ {}) or (EI_list ≠ {}) do
    no_issued := 0; \* Number of inst issued simultaneously at this time *\
    candidates := EI_list;
    if (EI_list = {}) curr_time := advance_clock(YES, YES, NO, curr_time);
    else if (lookahead = YES) \* Use Instruction Lookahead Heuristics *\
        BestChoice := get_ready_instr(candidates); \* The inst with the highest *\
        \* priority in the candidates list for which there are sufficient resources *\
        if (BestChoice ≠ NULL)
            while candidates ≠ {} do
                NextInst := get_ready_instr(candidates);
                if (NextInst ≠ NULL) apply.lookahead(BestChoice, NextInst);
            end while
            if (BestChoice.rdy_time ≤ curr_time + issue_cost)
                issue_instruction(BestChoice); no_issued++;
                EI_list := EI_list - BestChoice;

    else
        do \* Alternative strategy without Instruction Lookahead *\
            \* Let same_ETC_list be the list of the highest ETC cost ready insts *\
            ∃ same_ETC_list ⊆ candidates, s.t. ∀ i ∈ candidates,
                ∃ v ∈ same_ETC_list, s.t. (v.ETC ≥ i.ETC);
            candidates := candidates - same_ETC_list;
            do \* Remove instructions without sufficient resources *\
                ∃ readyI ⊆ same_ETC_list, s.t. ∀i ∈ readyI,
                    find_avail_FU_resources(i, datapath, curr_time);
                    if (|readyI| ≤ spsclr_deg - no_issued) issue_all(readyI, no_issued);
                    else \* choose between insts in readyI list *\
                        inst_chosen := choosing_insts(readyI, no_issued);
                        EI_list := EI_list - {inst_chosen};

            while ((no_issued < spsclr_deg) and (same_ETC_list ≠ {}));
        while ((no_issued < spsclr_deg) and (candidates ≠ {}));
        if (no_issued > 0) curr_time += inst_issue_cycle;
        else curr_time := advance_clock(YES, YES, YES, curr_time);
    end if
    update_writeback(datapath);
    if (WI_list ≠ {})
        update_WI(WI_list); update_EI(WI_list);
end while
update_writeback(datapath);
```

Example 1 : *Resource Lookahead*

switch IICT
  case 0: *Choose schedule* $\{A_1, B, A_2\}$ *or* $\{B, A_1, A_2\}$;
    * *Either schedule is optimal* *\
  case $(0 \leq \mathrm{IICT} < \frac{1}{2} \cdot \mathrm{ICT}_A)$:
    if $(\mathrm{ICT}_B > 2 \cdot \mathrm{ICT}_A)$ *Choose schedule* $\{B, A_1, A_2\}$;
      * *Instruction B takes longer than the both* A_1 *and* A_2 *\
    else *Choose schedule* $\{A_1, B, A_2\}$;
    * *In other words, combine the resource requirements of* *\
    * *dependent instructions and schedule the instruction* *\
    * *according to the resource with the most work.* *\
  case $(\frac{1}{2} \cdot \mathrm{ICT}_A \leq \mathrm{IICT} < \mathrm{ICT}_A)$:
    if $(\mathrm{ICT}_B > 2 \cdot \mathrm{ICT}_A)$ * *then schedule* B *first (as before)* *\
      Choose schedule $\{B, A_1, A_2\}$;
    else * *schedule* A_1 *first* *\
      if $(\mathrm{ICT}_B < \mathrm{ICT}_A)$ *Choose schedule* $\{A_1, A_2, B\}$;
    else *Choose schedule* $\{A_1, B, A_2\}$;
  case $(\mathrm{ICT}_A \leq \mathrm{IICT})$:
    * *Schedule the instruction with the largest* ICT *first* *\
    if $\mathrm{ICT}_A < \mathrm{ICT}_B$ *Choose schedule* $\{B, A_1, A_2\}$;
    else *Choose schedule* $\{A_1, A_2, B\}$;
end switch;

Example 2 : *Without Resource Lookahead*

if $(\mathrm{IICT} = 0)$ *Choose schedule* $\{A_1, B, A_2\}$ *or* $\{B, A_1, A_2\}$;
  * *Again, either schedule is optimal* *\
else * *Simply schedule the instruction with the largest* ICT *first.* *\
  if $(\mathrm{ICT}_A < \mathrm{ICT}_B)$
    Choose schedule $\{B, A_1, A_2\}$;
  else if $(\mathrm{IICT} < \mathrm{ICT}_A)$
    Choose Schedule $\{A_1, B, A_2\}$;
  else *Choose schedule* $\{A_1, A_2, B\}$;

local memory. In these architectures, instruction costs are well defined and considered fixed. Usually the latencies reflect the most optimistic execution, e.g., the time of a cache hit rather than a cache miss. Traditional schedulers improve performance through reordering instructions to avoid pipeline stalls, e.g., by inserting independent instructions after loads to keep the CPU busy. The number of instructions inserted (in the best case) depends on this latency value. If the load instruction is delayed beyond the scheduler's estimate, then the processor will stall. However, if the latency is shorter, then the destination register of the load instruction will be tied up for longer and this may increase register pressure enough to cause unnecessary code spills. Both balanced scheduling and resource lookahead are computationally more expensive than the traditional list scheduling approach, and will not be considered further in this paper.

4 Results

In this section, the makespans of MAP schedules for a number of typical instruction DAGs (briefly described below) are compared with their optimal. (The optimal makespan is derived from an exhaustive search.)

BT3 - A Binary Tree with three levels.

BT3.5 - A Binary Tree with three and half levels.

BT4 - A Binary Tree with four levels.

DD - Diamond DAGs which are commonly found in the evaluation of partial differential equations.

DM - Dense matrix multiplication.

SM - Sparse matrix multiplication.

CC - Mix of Load, Store and ALU instructions with data dependencies.

CCL - A loop unrolled version of CC.

Min1 - This architecture contains the minimum resources - one ALU and one Memory Unit (MU) which both share a single write-back bus. The cycle times and latencies of the ALU and MU micro-operations are assumed to the same.

3bus - This architecture has an additional ALU and each of the three functional units has a dedicated write-back bus. (The microoperation cycle times and latencies are the same as Min1).

Min2 - Same as Min1, except that the microoperation costs of the ALU and MU are different and reflect realistic costs obtained from SPICE-level simulations.

The results for the MAP scheduling heuristic, both without and with instruction lookahead, are shown in Table 1. For each DAG, the number of valid schedules is recorded together with the optimal makespan for the given target architecture. The makespan generated by the heuristics together with its closeness to the optimal (recorded both as a percentage of the optimal (*% Diff*) and as a percentage of the difference between the best and worst makespans (*% of the Range*) are also included. It is assumed that sufficient registers are available and so code spilling could be avoided. This would normally be determined at the register allocation phase of the compilation and is not considered here.

The results look quite promising. In a majority of the cases for the **3bus** architecture, the heuristic can find an optimal solution (only in the case of SM is instruction lookahead required to reduce the makespan to optimal). However, the MAP scheduler does not seem to do as well on the **Min1** architecture (for BT3, BT3.5, BT4, CCL, DM and SM). The reason for the poorer makespans is due to a bottleneck on the write-back bus. It turns out to be better in some cases to stall the issue unit for a longer period of time than that assumed by instruction lookahead (the IICT), i.e. wait until a higher priority instruction becomes ready, because this stall time is hidden by the write-back bottleneck. Where the makespan is only slightly worse than the optimal, i.e. DM, the heuristic together with instruction lookahead is sufficient to find an optimal solution. In the case of the **Min2** architecture, BT3, BT3.5, BT4, and CCL are now optimal. This is because the relative delays of the microagents have reduced the bottleneck for the write-back bus. In the case of DM and SM, there is still interference between the instructions which result in sub-optimal executions. Instruction interference can be reduced by applying a post-pass re-ordering of the generated schedules, and this is the subject of a future paper.

Remember that these schedules are only optimal with respect to the instructions costs which have been assumed. In practice, these schedules may not be optimal for a particular execution of the program for the reasons discussed earlier. One could even expect that each run of the program would have a different optimal schedule. The stability of the schedules in light of variance in the resource delays needs further study.

Prgm DAG	MAP Arch	No. of Valid Schds	No. of Optimal Schds	The Optimal Mkspn	*The MAP Heuristic*			*MAP with Lookahead*		
					Make-span	% Diff	% of the Range	Make-span	% Diff	% of the Range
BT3	Min1	640	24	1105nS	1185nS	92.76%	75%	1185nS	92.76%	75%
BT3.5	Min1	230400	512	1505nS	1585nS	94.68%	85.71%	1585nS	94.68%	85.71%
BT4	Min1	21964800	529920	1785nS	1885nS	94.4%	85.71%	1885nS	94.4%	85.71%
DD	Min1	42	2	1325nS	1325nS	100%	100%	1325nS	100%	100%
DM	Min1	310160	200	1905nS	1925nS	98.95%	98.11%	1905nS	100%	100%
SM	Min1	46574	24	2085nS	2245nS	92.33%	81.81%	2265nS	91.37%	79.55%
CC	Min1	4	2	735nS	735nS	100%	100%	735nS	100%	100%
CCL	Min1	4032	4	945nS	1015nS	92.59%	88.89%	1015nS	92.59%	88.89%
BT3	3bus	640	72	1105nS	1105nS	100%	100%	1105nS	100%	100%
BT3.5	3bus	230400	128	1355nS	1355nS	100%	100%	1355nS	100%	100%
BT4	3bus	21964800	456960	1605nS	1605nS	100%	100%	1605nS	100%	100%
DD	3bus	42	2	1225nS	1225nS	100%	100%	1225nS	100%	100%
DM	3bus	310160	156	1645nS	1645nS	100%	100%	1645nS	100%	100%
SM	3bus	46574	46	2005nS	2035nS	99%	97.67%	2005nS	100%	100%
CC	3bus	4	2	735nS	735nS	100%	100%	735nS	100%	100%
CCL	3bus	4032	18	835nS	835nS	100%	100%	835nS	100%	100%
BT3	Min2	640	32	930nS	930nS	100%	100%	930nS	100%	100%
BT3.5	Min2	230400	704	1230nS	1230nS	100%	100%	1230nS	100%	100%
BT4	Min2	21964800	768768	1500nS	1500nS	100%	100%	1500nS	100%	100%
DD	Min2	42	2	570nS	570nS	100%	100%	570nS	100%	100%
DM	Min2	310160	120	1250nS	1280nS	97.6%	92.5%	1250nS	100%	100%
SM	Min2	46574	2	1180nS	1200nS	98.3%	95.9%	1190nS	99.15%	97.96%
CC	Min2	4	2	400nS	400nS	100%	100%	400nS	100%	100%
CCL	Min2	4032	2	550nS	550nS	100%	100%	550nS	100%	100%

Table 1: Measuring the optimality of the scheduling heuristics

The Scheduling Costs

	Worst-case Costs	Average-case Costs
Issue Cycle Time	3	2
ALU Instruction	4	3
Load Instruction	10	4
Store Instruction	6	3

The Schedule Based on Worst-Case Costs

Instruction Schedule	Execution Time using worst-case run-time costs	Execution Time using average-case run-time costs
LD	0 - 10	0 - 4
ALU	3 - 7	2 - 5
ALU	6 - 11	4 - 7
ST	10 - 16	7 - 10
ALU	13 - 17	9 - 12
Execution Times :	17	12

The Schedule Based on Average-case Costs

Instruction Schedule	Execution Time using worst-case run-time costs	Execution Time using average-case run-time costs
ALU	0 - 4	0 - 3
LD	3 - 13	2 - 6
ALU	6 - 10	4 - 7
ST	9 - 19	6 - 9
ALU	16 - 20	8 - 11
Execution Times :	20	11

Figure 2: The makespans of schedules based on worst- and average-case run-time costs

5 Discussion

5.1 Choice of instruction execution costs

Although the execution times of the same instruction might vary due to data-dependent delays, worst-, average- or even best-case figures for the execution times can be found on which the schedules could be based. When producing static schedules, the compiler has to use the delays of the FMs and the question arises as to which of the sets of figures to use. Figure 2 illustrates the simplified schedules for the CC test (obtained from [15]) based on worst-case and average-case costs and figures for the execution times of the instructions based on actual worst-case and average-case delays at run-time for these schedules. (The ratios of the delays for the two cases for the instructions realistically reflect actual behaviour on the asynchronous processor under study.) The figures reveal that given these ratios, using a schedule based on worst-case costs is better in practice. Using this approach a heuristic will always try to schedule an instruction, if possible, only when its operands are guaranteed to be available, thereby minimising any stalls. Note also that the schedule's correctness is not affected by the changes in instruction costs. Furthermore, given that a program's critical path may change with different executions (due to different data sets) and that the schedule is generated once, the compiler's choice of which costs to use is important. By basing the schedule on worst-case delays a lower bound on performance can be achieved.

5.2 Interaction between executing instructions

Reasons other than the ones just stated also contribute to the difficulty in predicting the global state of the micronet. In synchronous processors, the compiler can assume when scheduling a basic block that the datapath is idle and that all of the resources are available. This is a consequence of the fact that in synchronous pipelines, an instruction never affects the execution of other instructions. This is not necessarily the case in a micronet, since the execution times of instructions might vary for the following reasons: only a partial ordering is employed between instructions (i.e. it is not necessary for the previous instructions to have completed their execution before successive ones); instructions compete for shared resources, e.g. the write-back bus; during execution instructions might interfere with each other. Therefore, the state of the resources at any particular time cannot be predicted accurately at compile-time. But this information is indeed available at run-time in the issue unit of the micronet. This could be used to dynamically tune (i.e. allow out-of-order instruction issues) the static schedule by the control unit. This requires identifying an instruction which can be executed immediately (easily achieved using the control acknowledgement signal scoreboarding mechanism), and checking that the instruction is independent of earlier ones in the instruction buffer. Although the latter may be expensive to perform, the task can be made simpler with assistance from the compiler by using a concurrency bit.

6 Conclusions

The MAP approach efficiently combines aspects of well-known architectural styles. In dataflow architectures, the instructions are issued as soon as their operands are available. This is achieved dynamically in hardware which incurs not insignificant run-time (book-keeping) costs. As in RISC architectures, code scheduling is done statically, but additionally instruction issue (and even possibly the instruction schedule) is fine-tuned dynamically to take advantage of run-time characteristics as in the data-flow model. In a sense, at the instruction-level MAP follows the classical von Neumann style, whereas at the level of microagents it is more in the character of dataflow architectures.

The micronet model exposes temporal and spatial concurrency in the datapath, with fine-grained resources now being visible to the compiler. This model subsumes the micropipeline model which only exploits temporal concurrency in the datapath and the scheduling methods described here can be equally applied to micropipeline-based processors.

Code scheduling (on ILP architectures) and machine-dependent optimisations have a significant impact on program performance. It is the task of the compiler to schedule instructions such that these resources are efficiently utilised. The instruction schedule is devised based on a computational model of the target architecture. For synchronous architectures the model is simple; in contrast, an asynchronous model is necessarily less accurate for the reasons discussed earlier. However, initial studies have shown that these factors do not significantly hinder a compiler's ability to schedule code efficiently. Worst-case instruction execution times have been considered and where the resulting schedule is treated as a first pass one. The interference between the instructions can be reduced by applying post-pass optimisations, and this is being currently investigated. The instructions could then be dynamically reordered at run-time to fine-tune this schedule by taking advantage of actual run-time costs. In conclusion, preliminary studies have shown that a micronet-based asynchronous processor architecture does present a suitable target for an ILP compiler.

Acknowledgements

V. Rebello was supported by a postgraduate studentship from the U. K. Engineering and Physical Sciences Research Council (EPSRC). This work was partially supported by a grant from EPSRC entitled *Formal Infusion of Communication and Concurrency into Programs and Systems* (Grant Number GR/G55457).

References

[1] T. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1978.

[2] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello. Micronets: A model for decentralising control in asynchronous processor architectures. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 190–199, London, UK, May 1995. IEEE Computer Society Press.

[3] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215, Leuven, Belgium, August 1994. Elsevier Science Publishers.

[4] D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In P. Dowd and E. Gelenbe, editors, *Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 100–105, Durham, NC, USA, January 1995. IEEE Computer Society Press.

[5] J. Baxter and J. H. Patel. The LAST Algorithm: A heuristic-based static task allocation algorithm. In *The Proceedings of the 1989 International Conference on Parallel Processing*, pages 217–222, 1989.

[6] E. Brunvand. The NSR processor. In *The Proceedings of the Hawaii International Conference on System Sciences.* IEEE Computer Society Press, January 1993.

[7] E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.

[8] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta. Informatica*, 1:200–213, 1972.

[9] I. David, R. Ginosar, and M. Yoeli. Self-timed architecture of a reduced instruction set computer. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[10] Mark E. Dean. *STRiP: A Self-timed RISC Processor.* PhD thesis, Stanford University, July 1992.

[11] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[12] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10, Grenoble, France, September 1993.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[14] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, December 1992.

[15] J. Hennessy and T. Gross. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[16] J-J. Hwang, Y-C. Chow, F. D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[17] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, November 1984.

[18] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *SIGPLAN Notices*, 28(6):278–289, June 1993. *Proceedings of the ACM Conference on Programming Language Design and Implementation.*

[19] S. J. Kim and J. C. Brown. A general approach to mapping of parallel computation upon multiprocessor architecture. In *The Proceedings of the International Conference on Parallel Processing, Vol. III*, pages 1–8, 1988.

[20] S. Manoharan and P. Thanisch. Assigning dependency graphs onto processor networks. *Parallel Computing*, 17(1):63–73, April 1991.

[21] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, Cambridge, Mass., 1989. MIT Press.

[22] C. McCreary, A. A. Khan, J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. Technical Report CSE-93-07, Auburn University, Auburn, AL, 36849. USA., 1994.

[23] S. V. Morton, S. S. Appleton, and M. J. Liebelt. ECSTAC: A fast asynchronous microprocessor. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 180–189, London, UK, May 1995. IEEE Computer Society Press.

[24] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test of Computers*, pages 50–63, Summer 1994.

[25] C. H. Papadimitrou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, April 1990.

[26] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. The MIT Press, 1989.

[27] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, Sun Microsystems Laboratories Inc., April 1994.

[28] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[29] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.