Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures

D. K. Arvind and S. Sotelo-Salazar

Department of Computer Science, The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, Scotland.

Abstract. This paper addresses the problem of scheduling instructions in micronet-based asynchronous processors (MAP), in which the latencies of the instructions are not precisely known. A PTD scheduler is proposed which minimises true dependencies, and results are compared with two list schedulers - the Gibbons and Muchnick scheduler, and a variation of the Balanced scheduler. The PTD scheduler has a lower time complexity and produces better quality schedules than the other two when applied twenty-three loop- and control-intensive benchmark programs.

1 Introduction

There has been a revival of interest in the use of asynchrony, albeit in a restricted form known as self-timing, in the design of processor architectures. Asynchronous circuits offer some distinct advantages. Their power consumption is generally much lower compared to their synchronous equivalent. This is because at any time only parts of the asynchronous system are active as required, with the rest remaining in a quiescent state. Self-timed systems allow a modular approach to processor design whereby parts can be added and deleted with little impact on the rest of the system. These systems are also robust to environmental changes.

The feature which is of most interest to our work and which was first recognised in the Micronet model [1] is that asynchrony offers scope for fine-grain concurrency in the processor architecture. The micronet model exposes this feature naturally, and asynchronous architectures based on this model are better able to exploit instruction-level parallelism.

A micronet-based architecture is viewed as a network of typed functional units. These units operate concurrently and communicate asynchronously with the rest of the architecture. The functional units themselves can be described at different levels of abstraction. In this paper the architecture is composed of the following functional unit types: one or more Arithmetic Unit (AU), a Logic Unit (LU), a Memory Unit (MU) and a Branch Unit (BU).

The issue and execution of an instruction consist of a sequence of micro-operations involving the Issue Unit (IU), the Register Bank, and the appropriate functional unit. An instruction is issued when both its operands are available. Once the instruction has been issued, it runs to completion unless it is stalled due to contention for resources in the trajectory of the instruction at any one of these points: the read ports, the functional unit, the write-back port. The micronet model enables concurrent execution of the micro-operations of the different instructions in flight, and minimises the costs of instruction stalls due to resource contentions. The latency of the instruction depends on a number of factors: its type, the data on which it operates, and the contention for resources which depends on the mix of instructions.

This paper proposes a relatively inexpensive method for scheduling instructions within the basic block. The objective of the scheduler is to ensure the rapid issue of independent instructions, thereby minimising the number of stalls of the issue unit, and in reducing the contention for the functional units by enabling instructions of different types to be in flight at the same time. This is achieved by assigning penalties to data dependencies and successive instructions of the same type, and transforming the schedule by moving instructions to reduce the penalties. This results in a schedule in which dependent instructions are separated, and independent instructions of different types are issued in succession.

The next section describes the traditional list scheduling algorithms such as Gibbons and Muchnick and the Balanced schedulers.

2 Traditional scheduling heuristics

2.1 The Gibbons and Muchnick (GM) scheduler

This is a well-known example of a list scheduling algorithm proposed originally for scheduling instructions in pipelined architectures [2]. The algorithm selects the instructions to be scheduled from a directed acyclic graph, beginning at the roots. The instructions are selected for scheduling if all their immediate predecessors have been scheduled. These *ready* instructions are prioritised on the following basis: if possible, an instruction is scheduled that will not interlock with the one just scheduled; given a choice, an instruction will be scheduled which is most likely to cause interlocks with instructions after it. The complexity in the absence of any lookahead in the instructions is $\theta(n^2)$, where n is the number of instructions in a basic block.

2.2 The Balanced scheduler

The Balanced scheduler [3] was devised to take account of unpredictable memory access latencies. The idea is to compute weights for load instructions based on the number of available independent instructions. The instructions are scheduled as in a traditional list scheduler with independent instructions being distributed behind loads to buffer for unpredictable memory accesses. This idea is extended beyond the load instruction to all the instructions in the MAP architecture. The priority for ready instructions is based on a weighted sum of values derived from MAP tailored heuristics - 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 DAG; and the number of source registers which are freed should the instruction be scheduled which effectively takes account of the register pressure.

3 The "Penalise True Dependencies" (PTD) scheduler

The essence of this heuristic is to identify true data and resource dependencies and re-order, where possible, the instructions such that their detrimental effect is reduced. The schedule is allocated a penalty measure based on the number and type of these dependencies. A true consecutive data dependency is penalised by one which is treated as the base case. If the dependency is with a branch or load instruction then it is penalised more severely. The actual value depends on the relative latencies of the functional units as shown in Table 1.

Instructions with resource dependencies are treated in a similar manner. If there are say p functional units of Type A, q units of Type B and r units of Type C, then a sequence containing more than p consecutive instructions of Type A, or q of Type B, or r of Type C will incur penalties. This assumes that the latencies of the three types of FUs are approximately the same; the run-length of the instructions can be suitably amended to take account of different latencies. The algorithm to derive this measure has a complexity of $\theta(n)$.

Cases of dependencies	Consecutive instructions	Separated by one inst.
True dependency with a load inst.	3	1
True dependency with a branch inst.	2	0
Resource dependency within mem. inst.	1	0
Normal true dependencies	1	0

Table 1. Table of penalties for true data dependencies.

We next demonstrate the correlation between the penalty measure considering only the true data dependencies and the makespans of the schedules for the program in Figure 1. The target asynchronous architecture has three types of functional units: an arithmetic unit (AU), logic unit (LU) and the memory unit (MU). The latency values for the units ranged over an interval, as shown in Table 2, with a Gaussian distribution. The results from a stochastic simulator which exhaustively simulated all the schedules (24,192) and averaged the results over 20 runs are shown in Figure 2. This result is representative of simulations of other programs with different spread of latencies. We can observe the trend that the penalty measure increases in step with the makespans of the schedules. This should ideally be a strict monotonic function, but the overlaps between the schedules of neighbouring penalties are tolerable for the heuristic approach. A scheduler based on minimising the penalty measure is introduced in the next section.

```
main() {

  int i, j, n = 10;
    int x[10];

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      x[i] = x[i] * x[j];

}
```

```
L4.main:
        muli    $13,$9,4
        la      $14,$29,0
        addu    $15,$14,$13
        muli    $24,$9,4
        la      $25,$29,0
        addu    $11,$25,$24
        lw      $12,$11,0
        muli    $13,$10,4
        la      $14,$29,0
        addu    $24,$14,$13
        lw      $25,$24,0
        mul     $11,$12,$25
        sw      $11,$15,0
        addui   $10,$10,1
        slt     $12,$10,$8
        bt      $12,L4.main
```

Figure 1. C and MAP assembly code from our example.

Component type	Minimum latency	Maximum latency
Issue Unit (IU)	1.00 ns	2.00 ns
Input buses	2.00 ns	4.00 ns
Output buses	2.00 ns	4.00 ns
Arithmetic Unit (AU)	4.00 ns	8.50 ns
Logical Unit (LU)	2.00 ns	7.00 ns
Memory Unit (MU)	10.00 ns	20.00 ns

Table 2. Latencies values for the target architecture.

3.1 The PTD scheduler

The PTD scheduler works in two phases: in the first phase the contention for resources is minimised, and in the second phase consecutive data dependent instructions are separated.

In the first phase, the types of consecutive instructions are compared and instructions are moved, where possible, so that the overall penalty measure is reduced, such that the number of consecutive instructions of the same type is no greater than the number of functional units of that type.

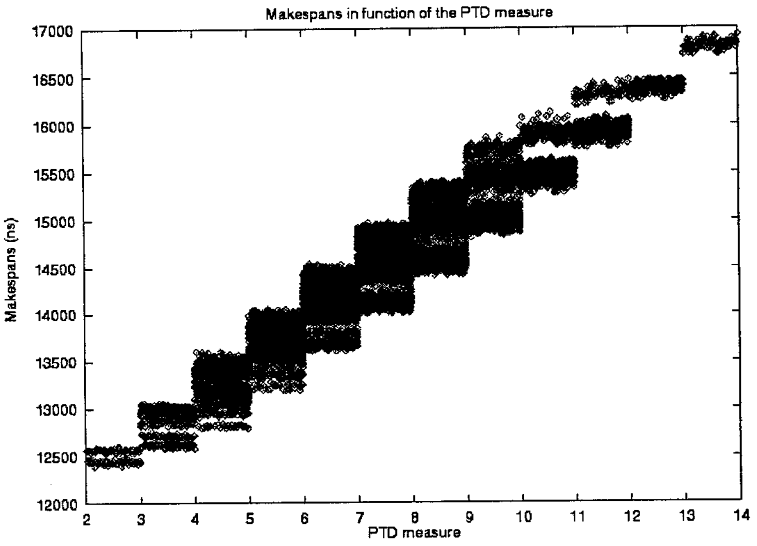

Figure 2. Execution distribution for the example.

In the second phase, the schedule is again scanned from start to finish, to identify consecutive data dependencies, and independent instructions are sandwiched in between them so that the overall penalty measure is reduced to zero or cannot be reduced any further due to the lack of suitable instructions. The details of the PTD scheduler are shown in Figure 3. The functions `PTD_arrange_left()` and `PTD_arrange_right()` traverse the schedule in both directions in search of independent instructions for insertion immediately after the penalised one. Two transformations are employed: a *swap* operation and a *move_ahead* operation and their use is illustrated in the following example.

Let l, m, n and o represent consecutive instructions in a schedule with a data dependency between n and o. This is represented by $n \dashrightarrow o$. The conditions for performing a $swap(m, n)$ transformation which eliminates (or reduces) the penalty to o, are the following:

- $m \parallel n$ (m is independent of n),
- $m \; ! \dashrightarrow o$ (meaning not producing a penalty) and
- $l \; ! \dashrightarrow n$

If the penalties go beyond consecutive instructions then in order to ensure that the penalty measure will be reduced after the *swap*, the necessary condition is that the sum of penalties before the movement is greater than the measure after the transformation is made.

The conditions for performing a $move\_ahead(x, n)$ (moves x ahead of n) to eliminate (or reduce) the penalty to o, are the following:

- $x \parallel a, \ldots, x \parallel l, \; x \parallel m, \; x \parallel n$,
- $x \; ! \dashrightarrow o$ and
- $x_{-1} \; ! \dashrightarrow x_{+1}$ where x_{-1} and x_{+1} are the instructions previous and following x, respectively.

```
void PTD_second_phase(dagnodes *root) {
  measure = PTD_measure(root, second_phase);
  if (measure > 0)
    do {
      node = root;
      last_measure = measure;

      while (node != NULL) {
        if (node -> PTD.penalised > 0)
          PTD_arrange_left (node);
        if (node -> PTD.penalised > 0)
          PTD_arrange_right(node);
        node = node -> next;
      }
      measure = PTD_measure(root, second_phase);
    } while (measure < last_measure && measure > 0);
}
```

Figure 3. The PTD scheduling algorithm - Phase 2.

Again to generalise the rules to allow a *move_ahead*, the sum of penalties before the insertion must be greater than the total number of penalties after the instruction has moved.

The conditions just outlined apply for the `PTD_arrange_left()` function which examines the left-hand side of the penalised instruction. The analogous conditions apply for the `PTD_arrange_right()` function but have been omitted for the sake of brevity. These conditions are sufficient to preserve the semantics of the program and reduce the PTD measure.

There will be cases where the only way to decrease the PTD measure of a schedule would be to replace a high penalty, i.e. load from memory, with a less expensive one, such as a "move register" instruction. So in terms of the penalty, one of 3 is reduced to 2 by moving an offending instruction, but the goal of reducing the overall measure is still accomplished.

The complexity of the PTD scheduler is $\theta(n\, e)$ where e is the number of penalties in the schedule. The worst case is one in which the schedule has at most $n-1$ consecutive dependencies (a pure sequential code) giving a complexity of $\theta(n^2)$ and the best case is $\theta(n)$. The linear-time complexity for the PTD scheduler is better than the $\theta(n^2)$ for the list scheduler [2] and $\theta(n^2\, \alpha\, n)$ [1] for the balanced scheduler [3].

4 Results

We next compare the quality of schedules produced by the Balanced, Gibbons and Muchnick (GM) and the PTD schedulers for a range of benchmarks which

[1] α is the inverse of the Ackerman function.

represent both loop-intensive (Livermore loops) and control-intensive categories of programs. These were compiled on the SUIF Compiler for the MAP target, but without any MAP-specific optimisations, and provided the same base schedule for the three schedulers under comparison.

The schedules were simulated on a discrete-event model of the MAP architecture. An architecture file describes the functionality and interconnection, and the spread of latencies as shown in Table 2. The distribution of latencies were chosen to best reflect the behaviour of the functional unit. The bimodal distribution for the Memory Unit captures the behaviour due to cache hits and misses. The distribution of the latencies for the Arithmetic Unit is based on the graph in Figure 4 in [4], and the distribution is uniform for the Logic Unit.

The simulation results presented in Figure 4, represent the average of five simulation runs for each program. They represent the percentage improvement with respect to the base case, i.e. the SUIF compiler output. The PTD scheduler outperforms the other two schedulers on both the control-intensive and loop-intensive programs.

When the number of AUs is increased from one to two (Fig. 5), we see a marked improvement in the schedules, but this tapers off when the AUs are increased further. This could be improved upon by scheduling instructions beyond the basic blocks. The favourable run-time complexity of the PTD algorithm makes this a practical proposition.

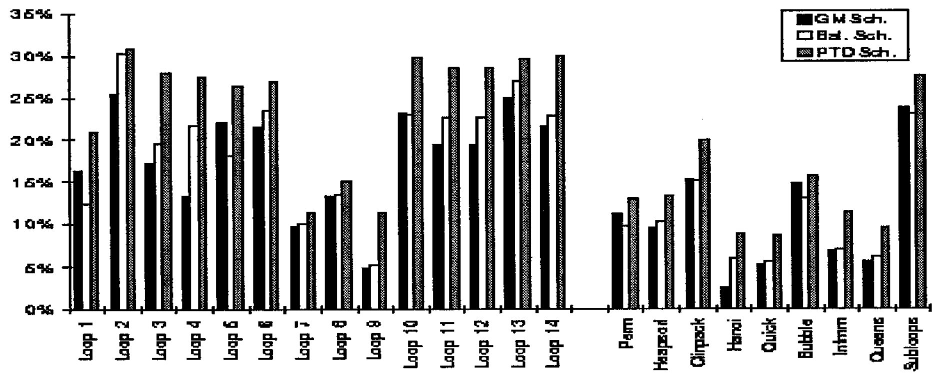

Figure 4. Average improvement for the whole set of benchmarks.

5 Conclusions

The PTD scheduler provides a simple yet effective method for scheduling instructions within basic blocks for programs running on MAP architectures. It has a better time complexity than the other two well-known list schedulers, and

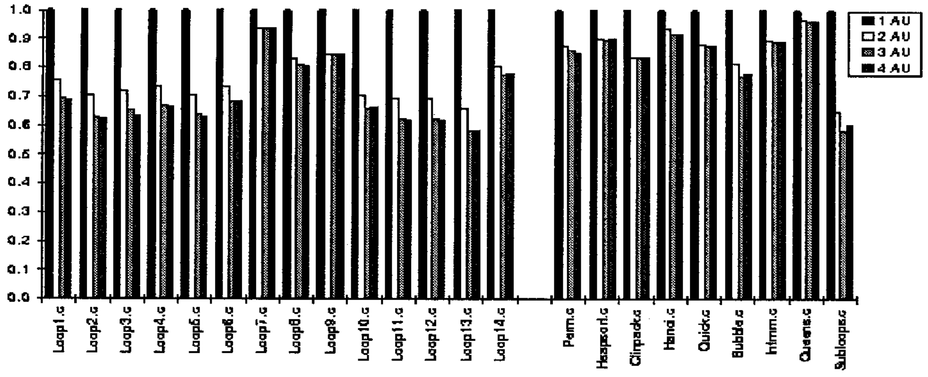

Figure 5. Ratio between the 1 AU and the other configurations.

the quality of the PTD schedules are better for a range of control- and loop-intensive benchmarks. The method reduces the stalls of the Issue Unit due to true data dependencies between instructions and enables better utilisation of the functional units by reducing the resource contention between instructions. The performance of the scheduler was investigated when the number of Arithmetic Units was scaled from 1 to 4. Future work will investigate the scheduling of instructions beyond the basic block boundaries for better utilisation as the functional units are scaled.

Acknowledgements

We would like to thank the members of the MAP group for useful discussions. S. Sotelo-Salazar was supported by a postgraduate studentship from the Science and Technology National Counsel in Mexico (CONACYT).

References

1. D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. *Proc. 3rd. International Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Belgium, August 1994, pp. 203-215.
2. P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proc. SIGPLAN 1986 Symposium on Compiler Construction*, SIGPLAN Notices, 21(7), July 1986, pp. 11-16.
3. D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *In ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 28(6), June 1993, pp. 278-289.
4. D. J. Kinniment. An evaluation of asynchronous addition. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, March 1996, pp. 137-140.