CP/MISF/$\infty$
A General Scheduling Technique for Treating Loops

Paulo Lorenzo$^1$, Munehiro Goto$^2$, and Arthur J. Catto$^1$

$^1$Department of Computer Science, UNICAMP
P.O. Box 6065, 13081-970, Campinas-SP, Brazil,
Tel: +55 (192) 39-8442, -7470 (fax),
Email: lorenzo@dcc.unicamp.br
catto@dcc.unicamp.br

$^2$Department of Computer Science, Gifu University,
1-1 Yanagido, Gifu, 501-11, Japan,
Tel and Fax: +81 (58) 293-2711,
Email: goto@goto.info.gifu-u.ac.jp

Abstract
Loops in the programs are a big source of parallelism, but their treatment can be very costly. There are several approaches to deal with loops, but just few are easy-handling. This paper proposes $\infty$ concept, a new way of seeing loops, and CP/MISF/$\infty$ scheduling technique. $\infty$ concept is a general way of dealing with loops in a large number of applications. Here we couple it with the well-known CP/MISF scheduling technique and propose CP/MISF/$\infty$ scheduling. In order to check up on the effectiveness of CP/MISF/$\infty$, we use the well-known Manchester Dataflow Machine (MDFM), one of the first data-driven machines designed and built. And, for the purpose of comparison, we evaluate four other scheduling techniques, FIFO, MISF, HLFNET, and CP/MISF, as well. The simulation results give a picture of the potentiality of CP/MISF/$\infty$ proposed here.

Keywords
Manchester Dataflow Machine, parallel architecture, parallel loops, parallel scheduling technique.
1 Introduction

An efficient task scheduling is one of the most critical points in a parallel system. Balancing the processing load among the Processing Units (PU) while managing the communication costs is a very difficult task. This difficulty increases many times as we consider to explore the latent parallelism that exists inside the loops.

The first step of dealing with loops in parallel systems was to process loops in parallel, that is, to pack the whole loop body into a task and process these tasks in parallel. Figuring out how to control the task size and how to estimate its processing time, for example, has been a researching line of this approach.

Running after more parallelism, new researches are dealing with processing the statements of the loop body in parallel. Basically these researches are heading two approaches. The first is the way to dedicate clusters of PUs to the loop tasks. And the second is to break up the loop body into some tasks and execute these tasks as ordinary ones under some scheduling control.

To pursue the latter approach, this paper introduces $\infty$ concept, a new way of seeing loops, and proposes CP/MISF/$\infty$ scheduling. This scheduling uses the well known CP/MISF [KN84] as main body and performs a static scheduling [CK88] of program tasks based on the priority labels given by the $\infty$ concept. In order to evaluate the $\infty$ concept and CP/MISF/$\infty$, the Manchester Dataflow Machine (MDFM) is here used as a target architecture. The choice of the MDFM is due to its desirable characteristics for this study as described below.

The MDFM is one of the first dataflow machines designed and built, and it is a well-known data-driven architecture with plenty of published information about [GKW85, GW83]. MDFM is a dynamic dataflow architecture [TBH82], which allows code sharing among loop steps and among calls within the same function\(^1\). Moreover, it works on extremely fine grain of computation and aims at the execution of program with very high parallelism, such as

---

\(^1\)The reentrant activation of functions in the dynamic dataflow approach is allowed by the concept of label or tag adopted. During the explanation of the MDFM in the next section, this concept is cleared up.
Meteorological Forecast.

In the MDFM, each task corresponds to an instruction. Such treatment maximizes the chance of parallelism detection in the programs, showing up all possible parallelism in the instruction level. And, the MDFM scheduling technique uses a listing whose entries are made based on the packet arriving order—first-in-first-out policy (FIFO).

The MDFM stands out as a perfect architecture for checking up on the effectiveness of the scheduling ways different from FIFO such as CP/MISF/∞. For the purpose of comparison, three other scheduling techniques, the well-known MISF, HLFNET [ACD74], and CP/MISF [KN84] are evaluated as well as FIFO and CP/MISF/∞.

In the next section we give a brief description of the Manchester Dataflow System, the FIFO implementation, and the simulator system developed and used here to evaluate the system performance. Then, the ∞ concept and CP/MISF/∞ are introduced in the section 3. The last three sections present the simulation results, the interpretation of the results and our conclusions, respectively.

2 Manchester Dataflow System

Some points on the MDFM necessary to understand the rest of the paper are described here. For detail, see [TBH82, GKW85, GW83].

MDFM is basically structured as a ring with a link of four main units connected to a host system via an Input/Output Switch Unit (see Figure 1). All units operate independently in a pipelined fashion and communicate through first-in-first-out queue structure. The Execution Unit, where there is a set of PUs and the task executions take place, is the only unit to work on a parallel philosophy.

The minimum execution unit in MDFM is called a task or a dataflow single instruction, whose input and output arguments are limited to at most two. Data are represented by tokens, composed of a value, a label and a destination address. These tokens are encapsulated in
packets which circulate around the ring. All packets have to pass by the Matching Unit (MU) in order to check out whether they have partners to match to or not.

In the next section, the scheduling technique used in the MDFM is sketched.

2.1 MDFM Scheduling Technique

The scheduling technique used in the MDFM is based on the simple first-in-first-out order list of the instruction arriving at the EU. It is recognized that the instruction availability order derived from FIFO does not necessarily correspond to the degree how each instruction contributes to the finish of the program. So in some cases, it may occur that instructions whose results are
only used after many cycles are executed earlier than the instruction whose results are needed right now in the program execution.

FIFO scheduling tries to enable as many paths of the program graph as possible, following the breadth-first execution order [RS87]. So there is a tendency to produce tokens whose partners will be available just after some cycles. Hence, these tokens have to stop to wait for their partners at the MU. This means an unnecessary utilization of the MU store resources and an undesirable concentration of wait and match operations [Lor].

2.2 MDFM Simulator

In order to analyze the MDFM characteristics a simulator is developed under the simplified system architecture assumptions. As our interest is in the machine functioning flow, this simulator works on the logical level of the program graph. There is no interest in the numerical magnitude of the results. The importance is both orders of the instruction executions and the token flow through the ring units. As in [GW83], the time to execute each instruction is supposed to be the same, hence the execution proceeds in discrete equal time steps.

All preliminary results obtained by our simulator are coincided with those based on the publications treating MDFM, specially in [GKW85, GW83]. So, it is recognized that our simulator has enough capability to evaluate the performance of the MDFM. For further information, see [Lor] which describes the simulator in detail but rather plainly.

3 CP/MISF/∞

Loops are identified in oriented graphs to the program by the existence of cycles (see Figure 2a). A cycle can be produced, for example, by a branch statement or a recursive call to a function. Making an analogy to the treatment of loops given by the demand model [TBH82], where the body of a loop is duplicated at each iteration of its execution, the graph of a loop could be viewed as an acyclic graph composed of an infinity number of copies of the cyclic
stretch (see Figure 2b). The real iteration number of some loops are decided at the execution time and not known in advance at the labeling phase. So we regard the iteration number as infinity and label \( \infty \) to the loop. This idea is called \( \infty \) concept.

![Figure 2: Global view of the \( \infty \) concept: a) a loop is viewed as b) an infinity number of copies of itself.](image)

In order to order the tasks based on the \( \infty \) concept, we define a polynomial, the \( \infty \) polynomial, as part of the task label. Based on that label we build a sorting rule that orders all buffers of the MDFM. Next we describe the \( \infty \) polynomial and the functioning scheme of CP/MISF/\( \infty \).

From leaves to root of the graph the tasks are labeled with the \( \infty \) polynomial. In each term of the polynomial the exponent means the degree of loop nesting, and the coefficient means the number of loops with the same nesting degree that have happened earlier down in the graph. A task labeled with the polynomial \( 2\infty^3 + \infty^1 \), for example, comes before two loops of nesting degree 3 and one loop of nesting degree 1, or it is within one of these loops. As we use CP/MISF as a body of CP/MISF/\( \infty \), the coefficient of term \( \infty^0 \) is filled with the level\(^2\) of the task in the graph. The level of each task is calculated by supposing that the graph is acyclic.

\(^2\)The level of a task is defined in the same sense as in [ACD74]. The level of a final task \( T_j \) is \( T_j \), and the level of a non-final task is defined to be the length of the longest path from that task to a final task.
And, together to this polynomial in each task label there is the number of output arguments the task produces. In the case of the MDFM, the number of output arguments can only be 1 or 2.

Hence, the task label by CP/MISF/$\infty$ has the form $<\infty$ polynomial, # output arguments$. The tasks $a$, $b$, $c$, and $d$ of the program graph in Figure 3, for example, have labels as $(\infty^3 + \infty^2 + 15, 1)$, $(\infty^2 + \infty + 15, 1)$, $(\infty + 1, 1)$, and $(1, 1)$, respectively. The task $a$ has the highest priority among these four tasks, for it has the highest $\infty$ polynomial, and for the analogous reason the task $d$ has the lowest priority. When tasks have the same $\infty$ polynomial, the one with bigger number of output arguments has the highest priority. And, when tasks have the same label, they are ordered randomly. Here for the latter case the tasks are ordered by the first-in-first-out policy.

The distinction between CP/MISF/$\infty$ and CP/MISF is the $\infty$ concept introduced here. By the $\infty$ concept we can fast prioritize tasks in the system taking account of whether they are part of loops or not.

As each instruction is a task itself in the MDFM, adding a label like the $\infty$ polynomial could overhead the communication system. However, if we just pack sequential stretches of the program code [LGC95], for example, into variable size tasks, this label would not cause a great overhead.

4 Evaluation Results

Let us consider the SUML program (see Figure 3). SUML is composed of two blocks, each of them with a call statement to the SUM program. And, the SUM call statement of block 1 is nested into a loop. As SUM program works by recursions, its available parallelism grows very fast.

Varying the seed of each branch statement in the SUML program, we can get different behavior and quite distinct latent performances. In order to avoid the low parallelism stretches
of the start and end of SUML execution, we use twenty copies of its code in parallel. We call this composition SUML20.

Table 1: Three set-up cases of SUML20.

<table>
<thead>
<tr>
<th></th>
<th># Branches</th>
<th>AP†</th>
<th># iterations of block I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block I</td>
<td>Block II</td>
<td>SUML20</td>
</tr>
<tr>
<td>Case 1</td>
<td>60</td>
<td>60</td>
<td>70,000</td>
</tr>
<tr>
<td>Case 2</td>
<td>60</td>
<td>60</td>
<td>35,297</td>
</tr>
<tr>
<td>Case 3</td>
<td>60</td>
<td>140</td>
<td>81,860</td>
</tr>
</tbody>
</table>

† Program average parallelism.

Table 1 describes three cases of SUML20. The blocks 1 and 2 of SUML20 are identical except for a branch statement. This branch statement causes the tasks to have different labels by CP/MISF/∞, which means different task priorities. But, this branch statement means nothing for the four other techniques, FIFO, MISF, HLFNET, and CP/MISF. Therefore, at first glance we might say that except for the case 2, where the branch statement is really effective, both cases 1 and 3 do not seem to be best for CP/MISF/∞ labeling, and that the case 1 seems to be best for HLFNET and CP/MISF.

Figures 4, 5, and 6 show the speedup curves of SUML20 executions of the cases 1, 2, and 3, respectively, with each of the following scheduling techniques: FIFO, MISF, HLFNET, CP/MISF, and CP/MISF/∞. In the speedup graphs (Figures 4, 5, and 6), CP/MISF/∞ is nicknamed C/M/∞.

In all three cases CP/MISF/∞ shows the best results. On the other hand, FIFO shows the worst results in all three cases. The other three techniques, MISF, HLFNET, and CP/MISF, show similar results among themselves in all cases. In the case 2, these three techniques show intermediate results between CP/MISF/∞ and FIFO. In the other two cases, 1 and 3, they show similar results to CP/MISF/∞.
The peculiar execution order breadth-first of FIFO scheduling technique brings about an overhead of the MU memory, which ends up to delay the system. On the other hand, the four other techniques, MISF, HLFNET, CP/MISF, and CP/MISF/∞, do a limited-breadth execution; go breadth-first until the machine becomes busy, and depth-first thereafter. For this reason, they show different results from FIFO.

In the case 2 of SUML20, where there is a straight and strong influence of a loop on
Figure 4: Speedup curves of SUML20, case 1, with different scheduling techniques.

Figure 5: Speedup curves of SUML20, case 2, with different scheduling techniques.
the program execution, only CP/MISF/$\infty$ keeps high performance. This result shows up the effectiveness of the $\infty$ concept. And, the results of the other two cases show up that to support CP/MISF with $\infty$ concept is quite effective to the performance improvement.

6 Conclusions

Exploring the parallelism of inside the loops can highly increase the overall performance of program execution. Machines that work on fine grain parallelism, such as MDFM, are able to break up the loop body, and explore the parallelism in the loop. However, not all machines are capable to cope with the dynamics of loops efficiently.

In order to improve the execution order of those parallel machines, in this paper we present an efficient scheduling technique for treating loops. Simulation results indicate that the $\infty$ concept of CP/MISF/$\infty$ can efficiently cope with loops. The simplicity and low overhead of CP/MISF/$\infty$ make it a good candidate for implementation on existing and future fine-grain
parallel machines.

The \( \infty \) concept can be easily coupled with almost any other scheduling technique. This versatility makes the \( \infty \) concept a general way of dealing with loops in a large number of applications.

Acknowledgements

The authors would like to thank to the Conselho Nacional de Pesquisa (CNPq) of the Government of Brazil and Nakashima Zaidan Association, Japan, for the financial support.

References


