

# Design criteria for applications with non-manifest loops

Omar Mansour    Egbert Molenkamp    Thijs Krol  
 University of Twente, Department of Computer Science  
 P.O. Box 217, 7500 AE Enschede, the Netherlands  
 Phone: +31 (0)53 4894178    Fax: +31 (0)53 4894590  
 E-mail: {mansour, molenkam, krol}@cs.utwente.nl

*Abstract*—In the design process of high-throughput applications, design choices concerning the type of processor architecture and appropriate scheduling mechanism, have to be made. Take a reed-solomon decoder as an example, the amount of clock cycles consumed in decoding a code is dependent on the amount of errors within that code. Since this is not known in advance, and the environment in which the code is transmitted can cause a variable amount of errors within that code, a processor architecture which employs a static scheduling scheme, has to assume the worst case amount of clock cycles in order to cope with the worst case situation and provide correct results. On the other hand a processor that employs a dynamic scheduling scheme, can gain wasted clock cycles, by scheduling the exact amount of clock cycles that are needed and not the amount of clock cycles needed for the worst case situation. Since processor architectures that employ dynamic scheduling schemes have more overhead, designers have to make their choice beforehand. In this paper we address the problem of making the correct choice of whether to use a static or dynamic scheduling scheme. The strategy is to determine whether the application possess non-manifest behavior and weigh out this dynamic behavior against static scheduling solutions which were quite common in the past. We provide criteria for choosing the correct scheduling architecture for a high throughput application based upon the environmental and algorithm-specification constraints.

*Keywords*— Non-manifest loop scheduling, variable latency functional units, dynamic hardware scheduling, self scheduling hardware units, optimized data-flow machine architecture.

## I. INTRODUCTION

A high throughput streaming application usually operates in an environment where continues processing of information takes place and lots of parallel operations, between consecutive input stream samples or data elements, have to be performed. The application usually performs a set of algorithmic operations on a continues stream of input data. The application itself usually consists of a set of interconnected algorithms. It is those set of algorithmic functions, their appropriate interaction and the underlying environment in which they have to operate, that form the character-

istics and the limitations of the application.

In order to achieve best results from such applications we need to build hardware architectures which are best suited for such applications, taking into account the environmental aspects in which they have to operate, characteristics and limitations of those class of applications.

## II. STREAMING APPLICATIONS AND THEIR ENVIRONMENTS

The environment in which an application is to operate has a tremendous influence on the choices designers have to take. We consider only streaming applications, as this paper is focused on researching such applications and architectures targeted for such applications. In a streaming environment, the life cycle of the application can be seen as consuming and processing input data-samples, and producing output samples. One restriction that is imposed on high throughput streaming applications, is that the implementation of the application has to be able to process the input stream within a bounded time interval, also known as the latency of the application. The term *highthroughput* in this paper refers to the fact that these type of applications have to perform multiple parallel operations within the time available between consecutive input stream samples. This means that standard processor architectures, which are based on the *von neumann* topologies are not suited here for, as they are not capable of performing multiple parallel operations. The data stream arrives at the inputs, of the application, with either a fixed input data-rate or variable input data-rate. If the data elements arrive with a fixed input rate we say that the system operates in a constant input-stream environment. Audio applications such MP3 players etc. form a typical constant stream application, as the data samples arrive with a constant speed in order to maintain the produced sound quality. On the other hand if the data elements arrive at variable time intervals, we say that the system operates in a variable input-stream environment.

A wireless communication medium can be considered as an example of a variable stream environment as the data packets or samples can be lost, due to the influences of noise or the medium in which the mobile is operating, or even due to transmission range constraints, and hence delivery is not guaranteed. Playing an MPEG movie from the hard disk of a personal computer could also be considered as a variable input-stream environment, since delivery of the data samples can be hindered by interrupts or other processes which are running on the personal computer, this is not the case in a real time operating system.

In the rest of this section we give a couple of definitions in order to explain some of the terminologies used within this paper.

*Definition 1 (stream)* An unbounded and ordered set of tokens  $S$ , where no two tokens  $p_i, p_j$  can exist at the same time is called a *stream* and is denoted by  $S$ . Hence  $S = -\infty \cdots p_{i-1}, p_i, p_{i+1} \cdots \infty$ .

**NOTE:** In practice the value of tokens range from small sizes such as samples produced from a high speed A/D converter to large blocks of data depending on the producing instrument or device.

*Definition 2 (streaming environment)* The environment in which an application operates is called a streaming environment, if the data tokens, the application operates on, arrive as a data stream and/or the application produces its output data as a stream.

*Definition 3 (variable streaming environment)* A streaming environment is called a variable streaming environment if the input or the output tokens do not have a constant inter-arrival/production time delay ( $\delta$ ).

*Definition 4 (constant streaming environment)* The data stream whether consumed or produced by an applications is called a constant stream, if the time delay *delta* between two consecutive tokens has a constant time value.

### III. NON-MANIFEST ALGORITHMS AND THEIR PROPERTIES

In this section we demonstrate the behavior of non-manifest loops, by providing a couple of real life algorithms. We show how to analyze the algorithm specifications, of a high throughput applications, and how to

recognize non-manifest behavior within them. Once non-manifest behavior is determined we can benchmark the algorithm and determine its execution latency distribution. We use this distribution in order to estimate the expected workload. Which is the parameter used in order to determine the required number of resources of the architecture.

Once non-manifest behavior has been determined and the application algorithms are bench-marked, the designers of the system are left with a number of choices which they can exploit. Designers can choose for solutions which modify the non-manifest behavior of the algorithm to a manifest behavior, hence providing the opportunity to build a processor architecture using static scheduling schemes or build a processor architectures which can exploit the non-manifest behavioral properties using dynamic scheduling schemes. Both solutions have their pros and cons, it is the applications at hand and the requirements of the system that dictate the appropriate solution to take. Modifying non-manifest behavior into manifest behavior is not possible for all algorithms and it involves algorithm transformation.

In this section we would like to demonstrate the properties of non-manifest algorithms by examining a couple of them. The first algorithm we analyze is the gcd() figure 1 function. We use this algorithm intensively as it demonstrates the characteristics of non-manifest loops and especially the analytic non-manifest loop as explained in [1]. By bench-marking this algorithm for all possible 16 bit input values we were able to determine the number of iterations needed for the worst case situation which is 23 iterations of the loop body, the best case situation which is 1 iteration of the loop body and finally the average number of iterations which was found out to be around 9 iterations.

We can say that an algorithm has non-manifest behavior if the execution latency for various input values is not a constant figure and in fact is dependent of the input data provided.

#### *The Gcd algorithm*

Figure 2 shows the input combinations which will result in the worst case and best case number of iterations. Figure 3 is the iteration distribution for all possible input combinations of length 16 bits. Although figure 3 gives us the distribution needed in order to determine the execution behavior of the gcd() function, it is not realistic to use this distribution in determining the number of resources that would be needed for

```

int gcd(int x, int y){
    int g;

    assert ((x>0) && (y>0));
    g = y;
    while ( x > 0 ){
        g = x;
        x = y % x;
        y = g;
    }
    return (g);
}

```

Fig. 1: Gcd algorithm

figure

x	y	g
28657,	46368,	46368
17711,	28657,	28657
10946,	17711,	17711
6765,	10946,	10946
4181,	6765,	6765
2584,	4181,	4181
1597,	2584,	2584
987,	1597,	1597
610,	987,	987
377,	610,	610
233,	377,	377
144,	233,	233
89,	144,	144
55,	89,	89
34,	55,	55
21,	34,	34
13,	21,	21
8,	13,	13
5,	8,	8
3,	5,	5
2,	3,	3
1,	2,	2
0,	1,	1

gcd(46368, 28657) == 1,  
num iterations == 23

x	y	g
0,	4,	4

gcd(4, 32768) == 4,  
num iterations == 1

Fig. 2: (a) Worst case execution versus (b) Best case execution of a gcd algorithm for 16 bit integer values figure

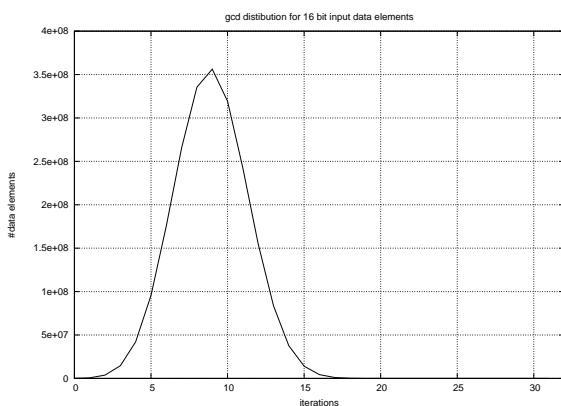


Fig. 3: Gcd() latency distribution in iterations figure

the final DFM architecture. The reason for this is that we can not assume that the whole set of 16 bit input values are provided as input during the execution of the application. In practice the application will be restricted by the output results of some other function or by the possible input values provided by the streaming environment. In order to know this information, one should bench mark the algorithm under the real life conditions where the application would be used and hence determine the expected work load of the application.

Another property which is of importance is the the input data stream throughput. The throughput together with the expected workload of the algorithm, give an indication to the amount of resources that are needed.

### The Cordic Rotation algorithm

The *cordic* algorithm is extensively used in the area of signal processing for calculating trigonometric functions including sine, cosine, magnitude and phase. *Cordic* revolves around the idea of "rotating" the phase of a complex number, by multiplying it by a succession of constant values. However, the multiplies can all be powers of 2, so in binary arithmetic they can be done using just shifts and adds, and hence no actual multiplier is needed. This property makes *cordic* suited for hardware processor implementations where no multipliers are available. The cordic rotation algorithm was chosen because it demonstrates another property, which can be exploited by designers in order to achieve high throughput. This property is: that the cordic rotation algorithm is a non-analytic non-manifest loop. This means that the loop converges to the correct answer and the more iterations are consumed the better the quality of the result is.

In figure 4 the cordic rotation algorithm that is bench-marked is provided. The algorithm stops when the c.y variable, line 9 of the code example in figure 4, is less than *APPROXZERO*. Figure 5 shows the error values made if we chose different values for *APPROXZERO* which is the stop criteria. In figure 6 we see the number of iterations consumed for each angle between 0 and 90 degrees for various values of *APPROXZERO*. Finally in figure 7 iteration distribution of various stop values is given. The designers of a high throughput application, which is to use this cordic algorithm, have now an the quality of the result as an extra criterium in the design process. They can chose to limit the number of iterations hence sacrificing the quality of the results if it is needed.

```

#define APROXZERO 0.000001

double rotate(complex c){
double z, p, tmpx, k;
int l;

z=0;
for (l=0; (fabs(c.y)>=APROXZERO)); l++){
p = phaseTab[l];
k = kTab[l];
tmpx = c.x;
if (c.y >= 0.0){
c.x += (c.y * k);
c.y -= (tmpx * k);
z += p;
} else {
c.x -= (c.y * k);
c.y += (tmpx * k);
z -= p;
}
}
return z;
}
    
```

Fig. 4: Cordic rotation algorithm

figure

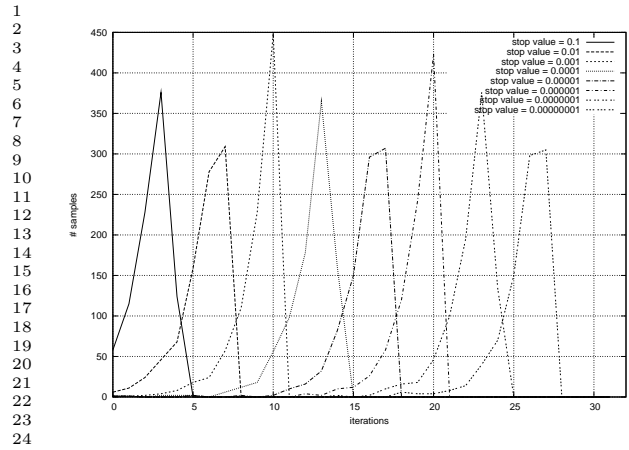


Fig. 7: Calculated cordic iteration distribution for various stop values

figure

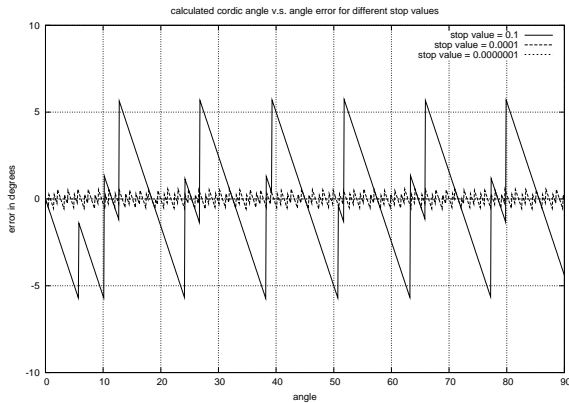


Fig. 5: Calculated cordic error v.s. stop value

figure

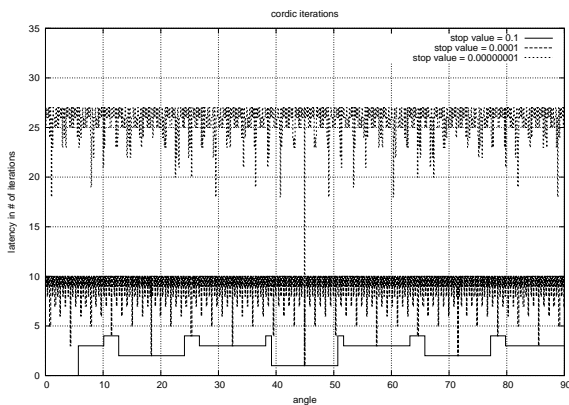


Fig. 6: Calculated cordic angle v.s. latency in iterations

figure

So the properties that were found are:

- The input and/or output throughput of the data stream
- The min, max and average execution latency of the algorithm
- The expected quality of the results
- Whether or not the algorithm poses non manifest behavior

Both the gcd() example of figure 1 and the cordic example of figure 4 possess non-manifest behavior as can be seen on their latency distributions in figure 7 and 3. The variation between the average case and the worst case execution latency for the gcd algorithm is more than 50%. This means that on average case we only require half the the number of resources that are needed for the static scheduling solutions which where used in the past.

#### IV. HARDWARE DYNAMIC SCHEDULING

Dynamic scheduling in the High2 DFM architecture figure 8 is performed within the execution unit itself.

The design of the execution unit mainly consists of an instruction table with memories for each index and their destination addresses. Each memory bank has an operand counter, which will count the number of operands of a certain index. Further there is an operand selector which determines which operands will enter the ready queue first. The ready queue stores the operands which are ready to execute. The operand selector makes use of an oldest frame pointer, this register points to the index of the memory holding the oldest operands in the system. The system also consists of a dispatcher/controller which performs the

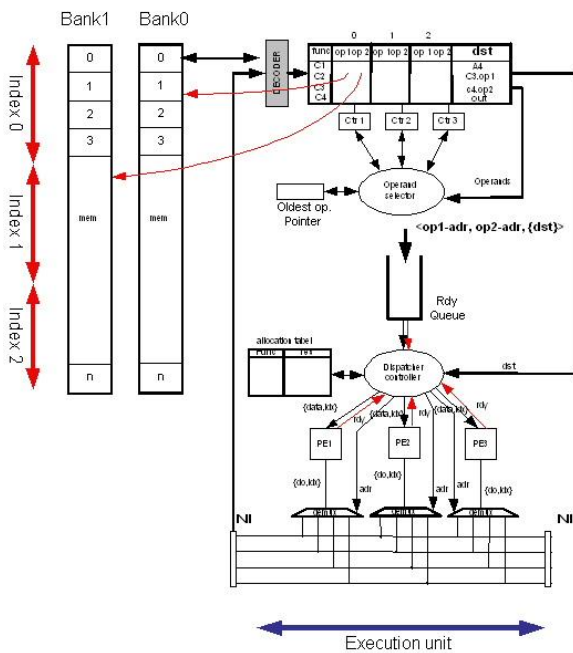


Fig. 8: High2 DFM execution unit figure

synchronization of the whole process, the resources which perform the actual computations and finally the destination-address multiplexors and their buses.

The process starts by searching the instruction table for operands which are ready to execute (left and right operands of a same time index which are available, we call this a match). The operand selector will select the matched operands based on their age, and hence the oldest operands are handled first by placing them first in the ready queue. At the start up of the system the oldest operand pointer points to the lowest index value. The operand counters are automatically incremented in a modulo fashion whenever a new operand match is found. Once the lowest operand counter overflows the oldest operand pointer will be incremented also in a modulo fashion. In other words if the oldest operand pointer points to the time index 2, and the operand match counter overflows the oldest operand pointer will point to time index 0. In this manner the system can keep track of the oldest operands in the system. The oldest operands that were written to the ready queue have a higher priority than other matched operands in the instruction

table and are handled first. In this fashion we can ensure correct execution of the system and that their will be no operand starvation, since operands within the system get older and at one time instance they will become the oldest operands and hence gain the highest priority. Operands of the same time index can enter the ready queue in an arbitrary fashion and hence there is no priority scheme for operands of the same age.

The queue has a special flag which indicates whether it contains ready operands or not. The dispatcher scans this flag and the ready-flags of the resources to see whether it can dispatch a new operand for execution or not. The ready flags of the resources indicate whether they are free or busy with a computation. The flags are needed as there is no means of indicating how long a computation of a non-manifest/variable-latency function will take. Once there is a free resource and there are ready instructions to be executed. The operands of that instruction will be dispatched to the free resource, and at the same time the dispatcher will write the resource number in its allocation table. Resources that have finished a computation will indicate this to the dispatcher using the same ready flag and at the same time write their output results to the multiplexors. The dispatcher will then select the correct destination buses using the information within the destination field of the instruction table and its allocations table. Once the destination address is known the address is provided to the multiplexors and the data will be placed on the bus. This data is either consumed by the instruction table of the destination resource or the output of the system. This process continuously repeats its self.

By examining the execution unit in figure 8 carefully we notice that the control process is mainly the controller dispatcher and operand selector. The rest of the unit such as the memory banks, ready queue operand table is just memory, A static scheduling approach will also require memory. This mainly means that the design of the execution unit might be larger than the of a static scheduling approach in terms of memory size. The expected control should not be that larger.

## V. CONCLUSIONS

In this paper we discussed the properties of non-manifest algorithms. Knowing that it is only possible, to build high throughput applications for non-manifest algorithms using static scheduling tech-

niques, by assuming the worst case execution latency and hence wasting a lot of clock cycles, depending off course on the latency distribution of the algorithm and also that the dynamic scheduling approach presented in [1] and [3] can gain those clock cycles at the cost of extra control overhead and a larger architecture design. Designers have to chose before hand which kind of scheduling scheme is to be used. By finding the correct criteria, designers of high-throughput applications which contain non-manifest loops can make the appropriate design choices. By knowing the input throughput and the anticipated work load of the environment and latency distribution of the input data samples for the real life situation. The correct choices can be made between static and dynamic scheduling approaches. Designers will have the opportunity to exploit the properties of non-manifest algorithms. These properties will lead to a more optimum hardware architecture in terms of the number of resources required.

#### REFERENCES

- [1] O.Mansour, S.Etalle, T.Krol, "Scheduling and Allocation of Non-Manifest Loops on Hardware Graph Models", PROGRESS Proceedings, 2001, ISBN 90-73461-26-x
- [2] Omar Mansour, Egbert Molenkamp, Thijs Krol, "The synthesis of a hardware scheduler for Non-Manifest Loops", EUROMICRO Symposium on digital system design, Dortmund, Germany 4-6 September 2002, ISBN 0-7695-1790-0
- [3] Omar Mansour, Egbert Molenkamp, Thijs Krol, "Minimum waste scheduling of dynamic variable-latency and non-manifest functional-units", PROGRESS Proceedings, 2002, ISBN 90-73461-34-0i