OPTIMISING MULTI-LOOP PROGRAMS FOR HETEROGENEOUS COMPUTING SYSTEMS

Y. M. Lam, J.G.F. Coutinho, W. Luk

Dept. of Computing,
Imperial College London.
{ymlam, jgfc, wl}@doc.ic.ac.uk

P. H. W. Leong

Dept. of Comp. Sci. and Eng.,
The Chinese University of Hong Kong.
phwl@cse.cuhk.edu.hk

ABSTRACT

This paper presents a method for optimising parallelisation and scheduling of task graphs containing representation of loops for implementation in heterogeneous computing systems with both software and hardware processors. The method integrates loop unrolling with task scheduling and determines the extent to which each loop should be unrolled to maximise performance, while meeting size constraints. A performance-driven strategy is proposed to find the best unrolling factor for each loop, such that the closer the match of run-time conditions and compile-time parameters, the higher the performance. Experimental results obtained using a speech recognition system show the proposed method outperforms an approach without unrolling by 2.1 times, and using the processing time of a 2.6GHz microprocessor as a reference, a speed up of 10 times can be achieved when compile-time and run-time parameters are matched, while the performance drops gradually when they are different.

1. INTRODUCTION

Computationally intensive tasks often involve iterative operations such as loops. Scheduling such loops properly e.g. parallelising the execution of various loop iterations, can potentially achieve better performance. Previous work on loop scheduling tends to address single loop only and assumes operations are synchronised by a global clock, so that a list scheduling strategy coupled with a simple model for estimating speed and area would suffice [11]. Various approaches are listed in Table 1 and summarized as follows:

- Path based [1],[7]: this approach divides a graph into sub-graphs, generating scheduling for each sub-graph independently. Since sub-graphs are treated independently, this approach may lead to sub-optimal solutions. It does not target an implementation with greater parallelism, since it does not cover multiprocessor systems.
- Modulo scheduling [2]: this approach generates a schedule for one iteration of a loop such that all iterations repeat the same schedule at regular interval, e.g. generating a software pipelining design.
- Loop transformation [8]: this approach converts a cyclic graph to acyclic directly using graph traversal algorithms such as depth first search. This approach does not analyze task dependency in different iterations which may result in less parallelism.
- Unrolling [13],[11],[10]: this is a common technique to generate an implementation with greater parallelism. It involves unrolling a loop and extracting parallel tasks from different loop iterations. However, current works focuses on parallelising a single loop.
- Dynamic scheduling [9]: this approach schedules tasks in run-time making use of both online and offline parameters, run-time conditions such as loop condition is monitored dynamically. However, loop parallelisation is not address in this approach.

Heterogeneous computing systems containing microprocessors and dedicated hardware processors, e.g. reconfigurable hardwares, provide potentially more effective solutions than single microprocessor systems for many real time and embedded digital signal processing applications. While previous work has focused on parallelising a single loop [2],[8],[13],[11],[10], it is noted that reconfigurable hardware in a heterogeneous system is capable of supporting parallel execution of tasks; the challenge is to develop techniques for effective exploitation of this capability. Recent work has shown the promising performance of an integrated mapping/scheduling system with multiple neighborhood function strategy [3] and the advantages of combining mapping/scheduling process with loop unrolling [4], this work complements previous work by providing a method for optimising the unrolling factors for multiple loops, which may be mapped to reconfigurable hardware in a heterogeneous system such that the performance of the system as a whole is

Table 1. Some approaches to address mapping/scheduling.

references	approach	examples of applications	comments
[1] [7]	Path-based scheduling	GCD, counter, Filtering	multiprocessors system not addressed
[2]	Modulo scheduling	DCT, FFT	analyze one iteration, single loop
[8]	Loop transformation	random graphs	less parallelism, single loop
[13] [11] [10]	Loop unrolling	random graphs, FFT, solver equalizer	single loop unrolling
[9]	Dynamic scheduling	fractal generation	loop unrolling not addressed, single loop
this work	Multi-loop unrolling	speech system	global unrolling factors determining, coarse-grained, heterogeneous systems

optimised. Moreover, additional management tasks are included so that the resulting system will always behave as expected, and the closer the match of run-time conditions and compile-time parameters, the higher the performance. The novel aspects are as follows:

- A static mapping and scheduling technique, capable of handling cyclic task graphs, supports loop unrolling such that the number of iterations may not be known until run-time.
- A performance-driven strategy, combined with an integrated mapping/scheduling system with multiple neighborhood function strategy, to find the best unrolling factor for each loop.

2. MAPPING AND SCHEDULING

2.1. System overview

Figure 1 shows the approach used to find a best mapping/scheduling solution for an input task graph. Given a task graph and a target architecture specification including characteristic of processing elements and communication channel, tabu search is used to generate different mapping/scheduling solutions (neighbors) iteratively. For each solution, an overall processing time, which is the time to finish all tasks, is calculated and used as the cost to guide the search. The goal is to find a solution with minimum overall processing time.

2.2. Integrated mapping and scheduling

Given a set of tasks $TK = \{tk_1, tk_2, ..., tk_n\}$ to be executed and a set of task lists $PL = \{pl_1, pl_2, ..., pl_m\}$, where $pl_j = (as_{j1}, as_{j2}, ..., as_{jq})$ is an ordered task sequence to be executed by processing element pe_j, each task in pl_j will

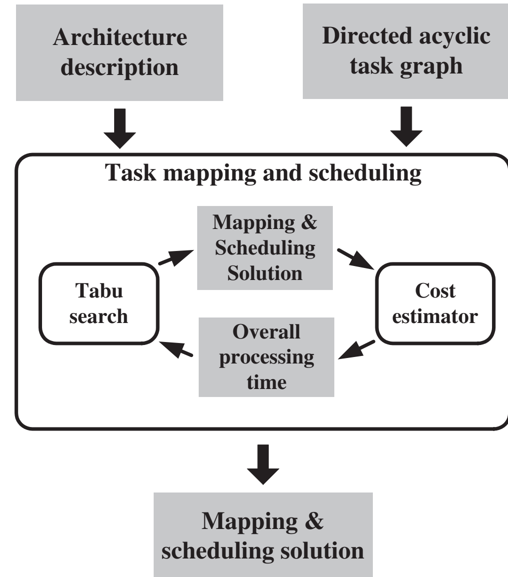

Fig. 1. Overview of mapping and scheduling solution generation.

be processed by pe_j in sequence when it is ready for execution, in other words, when all of its predecessors are finished. Task mapping and scheduling are thus integrated in a single step that deal with assigning tasks to task lists. A task assignment function is defined as A: $TK \rightarrow PL$, e.g. $A(tk_i) = as_{rq}$ denotes task tk_i being assigned to as_{rq} of list pl_r. This means that tk_i is the qth task to be executed by processing element pe_r. A mapping/scheduling solution is characterized by assignments of all tasks to processing elements, i.e. for every task $tk_i \in TK$, $A(tk_i) = as_{rq}$ for a $pl_r \in PL$. In this work, a clustering technique is further

integrated with mapping and scheduling process which will be introduced in the next Section.

2.3. Multiple neighborhood functions

The basis of heuristic search techniques is neighborhood search, which starts with a feasible solution and attempts to improve it by searching its neighbors, i.e. solutions that can be reached directly from the current solution by an operation called a move. Tabu search keeps a list of the searched space and uses it to guide the future search direction; it can forbid the search moving to some neighbors. In our proposed tabu search technique with multiple neighborhood functions strategy, after an initial solution is generated, two neighborhood functions are used to generate neighbors simultaneously. If there exists a neighbor of lower cost than the best solution so far and it cannot be found in the tabu list, this neighbor is recorded. Otherwise a neighbor that cannot be found in the tabu list is recorded. If all the above conditions cannot be fulfilled, a solution in the tabu list with the least degree, i.e. a solution being resident in the tabu list for the longest time, is recorded. If the recorded solution has a smaller cost than the best solution so far, it is recorded as the best solution. The searched neighbors are added to tabu list and solutions with the least degree are removed. This process is repeated until the search cannot find a better solution for a given number of iterations.

Given that s denotes current mapping/scheduling solution, the following two neighborhood functions $NFx(s)$, which have been shown to provide best performance [3], are used in this work:

NF1: ***guided-relocation***

(1) Randomly select a task at list pl_p position as_{pq}.
(2) For all lists $pl_j \in PL$ and positions as_{jk}: relocate task (pl_p, as_{pq}) to (pl_j, as_{jk}) which yields lowest cost.
(3) Relocate largest data flow parent of task (pl_p, as_{pq}) to a position in pl_j which yields lowest cost.

NF2: ***guided-swap***

(1) Randomly select a task at list pl_p position as_{pq}.
(2) For all lists $pl_j \in PL$ and positions as_{jk}: swap task (pl_p, as_{pq}) with task (pl_j, as_{jk}) which yields lowest cost.

Based on the observation that mapping tasks with large data flow to the same processing element can potentially reduce data transfer overhead, a clustering technique which tries to group tasks with large data flow together and allocate them to the same processing elements, is proposed [12]. However, clustering, mapping, and scheduling are solved separately, which may result in sub-optimal solutions. our approach provides an improvement that integrates the clustering with mapping/scheduling steps, by introducing neighborhood function $NF1$. So after relocating a task, the system tries to move the largest data flow parent, which has the largest data flow among all parents of the current task, to the same processing element. If such move yields lower cost, this move is accepted and a new neighbor is generated, otherwise, it is discarded.

2.4. Cost function

As mentioned before, overall processing time is used as the cost to guide the heuristic search. This is the time for processing all tasks using the reference heterogeneous computing system and includes data transfer time. The processing time of a task tk_i on processing element pe_k is calculated as the execution time of tk_i on pe_k plus the time to retrieve results from all of its predecessors. The data transfer time between a task and its predecessor is assumed to be zero if they are assigned to the same processing element.

A speed up coefficient is defined to measure the quality of a mapping/scheduling solution, it is calculated as the processing time using a single CPU divided by the processing time using the heterogeneous computing system:

$$SU = \frac{processing\ time_{single\ CPU}}{processing\ time_{Reference\ system}} \quad (1)$$

It is noted that tasks cannot be randomly moved to any location due to data dependency, i.e. a task cannot be moved to a location such that it will execute prior to its predecessor. To avoid generating infeasible solution, our approach inflicts a huge penalty on the cost of the reference system if such move is infeasible.

3. MULTI-LOOP PARALLELISATION

3.1. Single loop parallelisation

In order to parallelise an application, an unrolling based approach is used. The basic strategy is unrolling a loop and process different iterations of the loop body in parallel. Given an unrolling factor, which is defined as the number of iterations to be unrolled, the parallelisation process is shown as follows:

1. Unroll the loop for a given unrolling factor.
2. Construct a new task graph by treating each unrolled iteration as a new task.
3. Generate a management task to synchronise results produced by different iterations, and insert this task into task graph.
4. Generate a complete mapping/scheduling solution using the strategy described in Section 2.

If there is only one loop in an application, it is obviously that this loop will be selected for unrolling. However, if an application contains various loops, a questions is raised: how to determine the unrolling factor for each loop? This is addressed in the following section.

3.2. Determination of unrolling factors

The advantage of considering unrolling of all loops globally is that tasks in different iterations of various loops can potentially be executed in parallel. An iterative search using a performance driven strategy is proposed to find the best unrolling factor for each loop (Algorithm 1), it is noted that this strategy is applicable to applications containing multiple parallel loops or nested loops. Initially, unrolling factor UF_i of all loops are set to 1. For each loop i, a speed up improvement ISU_i is then calculated which is the speed up difference before and after unrolling. A loop with maximum ISU is selected and the unrolling factor of this loop is increased by one, and the unrolled iterations are pre-mapped to FPGA, forming a partial mapping solution. Subsequently, a complete mapping/scheduling solution is generated using the technique described in Section 2. This process is repeated until there is not enough FPGA resources for mapping tasks.

If an estimated loop count for a particular loop is provided by user or a loop count is pre-defined in compile-time, this loop count is used as the maximum iteration number for unrolling, i.e. the maximum value for the unrolling factor UF_i. A loop is fully unrolled if the unrolling factor UF_i reach this maximum.

Algorithm 1 Search of best unrolling factor

1: set all unrolling factors $UF_i \Leftarrow 1$
2: $currentSU \Leftarrow 0$
3: $usedFpgaResource \Leftarrow 0$
4: **while** $usedFpgaResource < totalFpgaResource$ **do**
5: $maxISU \Leftarrow 0$
6: **for all** loop LP_i **do**
7: $curUF \Leftarrow UF_i + 1$
8: Unroll LP_i for $curUF$ iterations
9: Map unrolled iterations of LP_i to FPGA
10: Generate a complete mapping/scheduling MS_i
11: Calculate speed up SU_i for MS_i
12: $ISU_i \Leftarrow SU_i - curSU$
13: **end for**
14: find loop max_i with maximum ISU
15: $UF_{max_i} \Leftarrow UF_{max_i} + 1$
16: Unroll loop LP_{max_i} for UF_{max_i} iterations
17: Map unrolled iterations of LP_{max_i} to FPGA
18: $BestMS \Leftarrow MS_{max_i}$
19: $currentSU \Leftarrow SU_i$
20: Update $usedFpgaResource$
21: **end while**
22: **return** $BestMS$ and UF

3.3. Management task

One of the problems introduced after unrolling is data synchronisation: since results are produced by unrolled iterations in parallel, they need to be organised in correct order (Figure 2). Another problem is loop count uncertainty, e.g. a loop is being unrolled n times, but the actual loop count at run-time may not be a multiple of n. That means some unrolled iterations have produced useless results and these results need to be discarded. We propose a strategy to handle these problems by generating a management task to handle data synchronisation. The management task is treated as a task and inserted into the task graph, which is then presented to the mapping/scheduling tool. For loops without data dependency between iterations, the following pseudocode shows an example of data synchronisation for unrolling three iterations.

```
for (i=0; i<(M-1); i++) {
    rst[i*3] = d0[i];
    rst[i*3+1] = d1[i];
    rst[i*3+2] = d2[i];
}
tc = M * 3 - N;
switch(tc) {
    case 0:
        rst[(M-1)*3] = d0[M];
        rst[(M-1)*3+1] = d1[M];
        rst[(M-1)*3+2] = d2[M];
        break;
    case 1:
        rst[(M-1)*3] = d0[M];
        rst[(M-1)*3+1] = d1[M];
        break;
    case 2:
        rst[(M-1)*3] = d0[M];
        break;
}
```

where M is the actual count of executing the unrolled loop, N is the original loop count. $d0$, $d1$ and $d2$ are the results produced by the unrolled three iterations respectively. rst is the original array to store results.

If there is data dependency between iterations, the functionality of the management task is selecting the correct result from unrolled iterations:

```
tc = M * 3 - N;
switch(tc) {
    case 0:
        rst = d2;
```

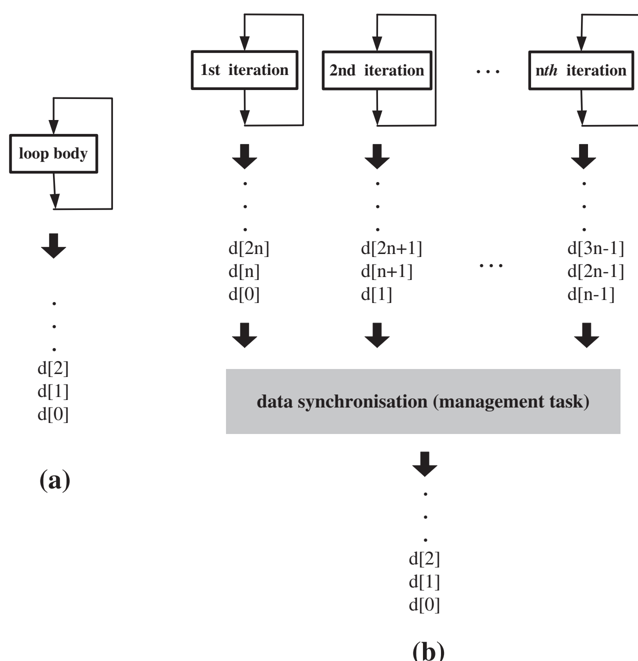

Fig. 2. Unroll loops without data dependency between iterations: (a) Original loop. (b) Unrolled loop and data synchronisation.

```
            break;
        case 1:
            rst = d1;
            break;
        case 2:
            rst = d0;
            break;
    }
```

By generating these management tasks to handle data synchronisation, the generated mapping/scheduling solution does not require the designer to know run-time conditions accurately. Consider the case when a user specifies an estimated loop count at compile time: the loop is unrolled using this information and a mapping/scheduling solution is generated. If the estimated loop count matches the actual value at run-time, maximum performance can be achieved. However, if the loop count at run-time is different, the generated data management task can handle data synchronisation dynamically, which means the generated mapping/scheduling solution is still feasible. These management tasks can easily be implemented in software or in hardware state machines.

4. RESULTS

4.1. Experimental setup

The reference heterogeneous computing system used here contains one 2.6 GHZ AMD Opteron(tm) Processor and one Celoxica RCHTX-XV4 FPGA board with a Xilinx Virtex-4 XC4VLX160 FPGA. Both the FPGA board and the microprocessor are connected by HTX interface with a maximum data transfer rate up to 3.2 GB/s.

An isolated word recognition system [6] is selected as a case study which uses 12th order linear predictive coding coefficients (LPCCs), a codebook with 64 code vectors, and 20 hidden Markov models (HMMs), each has 12 states. One set of utterances from the TIMIT TI 46-word database [5] containing 5082 words from 8 males and 8 females are used for recognition. Profiling results on the AMD processor show that loops in vector quantisation (vq), autocorrelation (autocc) and hidden Markov model decoding (hmmdec) consuming largest CPU resource, which are 71.19%, 15.4% and 6.11% respectively.

4.2. Multi-loop unrolling

The proposed unrolling strategy is applied to find the best unrolling factor for each loop and generate the best mapping/scheduling solution for the speech system. In this experiment, it is assumed that the order of LPCC, size of codebook, number of HMMs and states are known at compile-time. It is found that vector quantisation is being unrolled for 3 iterations (vq1-3), autocorrelation process is fully unrolled for 12 iterations (autocc1-12). Hidden Markov model decoding contains nested loops, the inner loop is unrolled for 12 iterations (hmmdec1-12) which is equal to the number of HMM states, the outer loop is further unrolled for 2 iterations which means two HMM decoding processes are executed in parallel. The corresponding FPGA resource usage is shown in Table 2 and the operating frequency is 318.7 MHz. The speed up obtained after unrolling is 10 and the speed up obtained without unrolling is 4.7, where vector quantisation, autocorrelation and HMM decoding are executed in FPGA without unrolling. Hence an improvement of 2.1 times is obtained using the proposed unrolling technique (Table 3).

4.3. Run-time vs compile-time parameters

In the above experiment, mapping/scheduling solutions are generated based on a pre-defined LPCC order of 12. However, this value may be modified to cope with different circumstances at run-time. Using a mapping/scheduling solution generated with 12 LPCCs, Figure 3 shows the performance of this system for different run-time LPCC orders. It is found that maximum performance is achieved at 12 LPCCs, and the performance drops when the run-time

Table 2. FPGA resources of different speech processes, the total resource is calculated by counting two "hmmdec1-12".

Process	Resource (slice)
vq1-3	21819
autocc1-12	10272
hmmdec1-12	15948
Total slices used	63987
Total slices available	67584

Table 3. Performance comparison before and after unrolling.

	Speed up	Used FPGA resource
Without unrolling	4.7	14%
With multi-loop unrolling	10	94.7%
Improvement	212.8%	

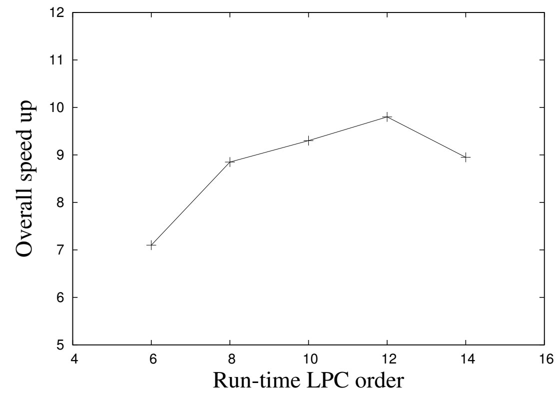

Fig. 3. Speed ups for different run-time LPCC orders, the compile-time LPCC order is 12.

LPCC order is different from compile-time value, but the proposed strategy can still provide a feasible system.

5. CONCLUSIONS

A static loop unrolling based task graph mapping/scheduling technique is presented. It is capable of generating generic mapping/scheduling solutions which do not require designers to know the run-time conditions accurately. A performance driven based strategy is proposed to find the best unrolling factor for each loop. Experimental results obtained using a speech analysis system show that the proposed unrolling strategy outperforms an approach without unrolling by 2.1 times, and a speed up of 10 times is achieved. The more accurate the compile-time information about the run-time conditions, the higher the performance; however, a feasible solution can always be produced.

6. REFERENCES

[1] R. Camposano. Path-Based Scheduling for Synthesis. *IEEE Transactions on Computer-Aided Design*, 10(1):85–93, Janurary 1991.

[2] A. Hatanaka and N. Bagherzadeh. A Modulo Scheduling Algorithm for a Coarse-Grain Reconfigurable Array Template. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[3] Y. M. Lam, J. G. F. Coutinho, W. Luk, and P. H. W. Leong. Mapping and Scheduling with Task Clustering for Heterogeneous Computing Systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 275–280, 2008.

[4] Y. M. Lam, J. G. F. Coutinho, W. Luk, and P. H. W. Leong. Unrolling-based loop mapping and scheduling. In *Proceedings of the International Conference on Field Programmable Technology*, 2008, accepted.

[5] LDC. *http://www.ldc.upenn.edu*.

[6] L. Rabiner and B. H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall PTR, 1993.

[7] M. Rahmouni and A. A. Jerraya. Formulation and Evaluation of Scheduling Techniques for Control Flow Graphs. In *Proceedings of the Design Automation Conference*, pages 386–391, 1995.

[8] F. E. Sandnes and O. Sinnen. A New Strategy for Multiprocessor Scheduling of Cyclic Task Graphs. *International Journal of High Performance Computing and Networking*, 3(1):62–71, 2005.

[9] H. Styles, D. B. Thomas, and W. Luk. Pipelining Designs with Loop-carried Dependencies. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 255–262, 2004.

[10] P. Sucha, Z. Hanzalek, A. Hermanek, and J. Schier. Efficient FPGA Implementation of Equalizer for Finite Interval Constant Modulus Algorithm. In *Proceedings of the International Symposium on Industrial Embedded Systems*, pages 1–10, 2006.

[11] M. Weinhardt and W. Luk. Pipeline Vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):234–248, February 2001.

[12] T. Wiangtong, P. Y. K. Cheung, and W. Luk. Hardware/Software Codesign: A Systematic Approach Targeting Data-intesive Applications. *IEEE Signal Processing Magazine*, 22(3):14–22, May 2005.

[13] T. Yang and C. Fu. Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):608–622, June 1997.