Unrolling-based loop mapping and scheduling

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 Computer Science and Engineering,
The Chinese University of Hong Kong.
{phwl}@cse.cuhk.edu.hk

Abstract

This paper presents an loop unrolling based mapping and scheduling strategy to maximum the parallelism of an application described as task graph targeting on a heterogeneous computing systems. Loops are statically unrolled using compile-time parameters and dynamic tasks are generated to handle run-time conditions, 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

Heterogeneous computing systems containing software processors (e.g. microprocessors) and hardware processors (e.g. reconfigurable hardware) provide potentially more effective solutions than single microprocessor systems for many real time and embedded digital signal processing (DSP) applications. As we know, computational intensive parts of such applications are usually iterative operations such as loops, the problem of generating an implementation for heterogeneous computing systems given a DSP application, such that the hardware resource is fully utilised and parallelism is maximised, is still difficult. Various approaches have been proposed to address the scheduling problem for loop, such as scheduling based on control path [1],[6], modulo scheduling [2], loop transformation [7] and unrolling [11],[10],[9], dynamic scheduling [8].

While previous work has focused on paralleling a single loop [2],[7],[11],[10],[9], it is noted that a heterogeneous system containing reconfigurable hardware is capable of supporting parallel execution of tasks; the challenge is to develop techniques for effective exploitation of this capability. Recent work [3] involving an integrated mapping/scheduling system with multiple neighbourhood function strategy shows promise in mapping acyclic task graphs into heterogeneous systems. This work addresses multiple loops parallelisation for task graphs by providing a solution with additional management tasks so that the resulting system is functionally correct.

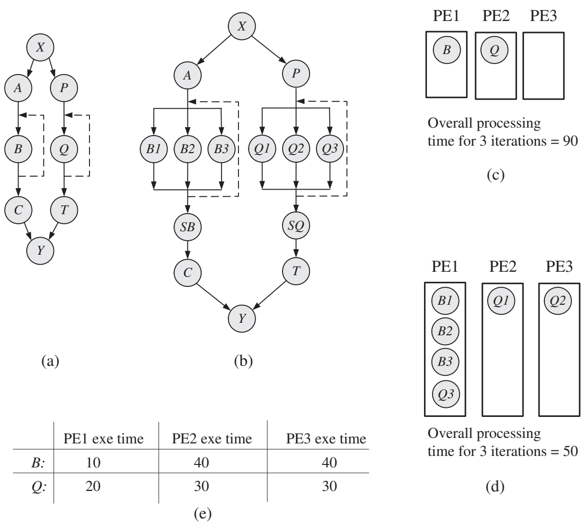
2 Methodology

2.1 Unrolling-based loop mapping and scheduling

Given an application described as task graph and unrolling factor for each loop, each loop is unrolled and management task, which is used for data synchronisation (Section 2.3), is inserted to form a new task graph. Feedback edges of loops are then removed using depth first search [7], such that the task graph is converted into directed acyclic task graph. An integrated approach [3] is finally used to generate a mapping/scheduling solution for the targeting heterogeneous computing system. In this way, parallelism between different loops and different iterations is fully explored by the mapping and scheduling system.

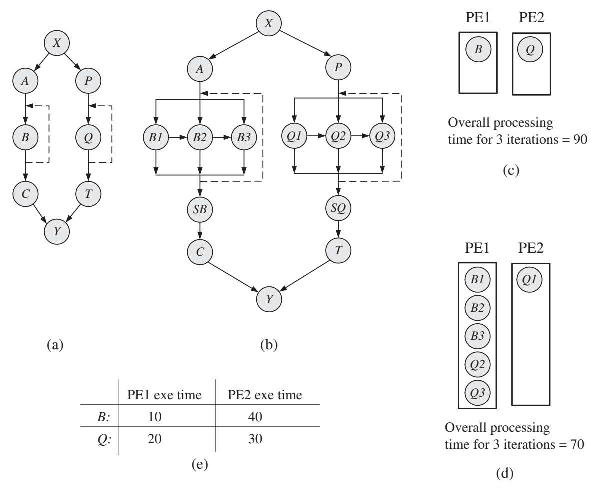
2.2 Loop unrolling

The advantage of considering unrolling of all loops globally is that tasks in different iterations of various loops can be potentially executed in parallel, i.e. a better scheduling can be found after unrolling. Figure 2.1 shows the unrolling example of two loop without data dependency between iterations. In the original graph, B and Q are loop bodies which can be single tasks or sub-graphs, $B1$, $B2$ and $B3$ are the 3 unrolled iterations of task B, and task Q is unrolled as $Q1$, $Q2$ and $Q3$ respectively. Before unrolling, B and Q are mapped to PE1 and PE2, hardware resource is not fully utilised, the processing time for 3 iterations using this mapping is 90 (Figure 2.1c). After unrolling, the first two


(a)
(b)
PE1
PE2
PE3
Overall processing time for 3 iterations = 90
(c)
Overall processing time for 3 iterations = 50
(d)
(e)

	PE1 exe time	PE2 exe time	PE3 exe time
B:	10	40	40
Q:	20	30	30

Figure 1. Example of loop unrolling when there is no data dependence between iterations: (a) Original graph. (b) Unrolling for three iterations. (c) Mapping and scheduling before unrolling, overall processing time for 3 iterations is 90. (d) Mapping and scheduling for unrolled loop, overall processing time is 50. (e) Execution time of tasks on different processing elements. Higher inter-loop and intra-loop parallelism is achieved by unrolling two loops.


(a)
(b)
PE1
PE2
Overall processing time for 3 iterations = 90
(c)
Overall processing time for 3 iterations = 70
(d)
(e)

	PE1 exe time	PE2 exe time
B:	10	40
Q:	20	30

Figure 2. Example of loop unrolling when there is data dependence between iterations: (a) Original graph. (b) Unrolling for three iterations. (c) Mapping and scheduling before unrolling, overall processing time for 3 iterations is 90. (d) Mapping and scheduling for unrolled loop, overall processing time is 70. (e) Execution time of tasks on different processing elements. A higher inter-loop parallelism is achieved by unrolling two loops.

iterations ($Q1$ and $Q2$) of Q are mapped to PE2 and PE3 respectively, other unrolled iterations are mapped to PE1, the processing time for this mapping is 50 (Figure 2.1d).

If a loop has data dependency between iterations, unrolling can still achieve higher parallelism. An example is shown in Figure 2 which contains two loops with data dependency between iterations. Before unrolling, task B is mapped to PE1 and Q is mapped to PE2, the overall processing time for 3 iteration is 90 (Figure 2c). After unrolling for 3 iterations, the first iteration of task Q (i.e. $Q1$) is mapped to PE2, and the remaining iterations can be executed in PE1, the overall processing time becomes 70 (Figure 2d). A better mapping/scheduling solution with higher inter-loop parallelism is thus obtained by unrolling.

2.3 Management task

One of the problem need to address after unrolling is data synchronisation. Since results are produced by different unrolled iterations in parallel, these results need to be sorted into correct order. The other problem is raised by loop count uncertainty, e.g. a loop is being unrolled n times, but the actual loop count may not be a multiply of n. That means the unrolled iterations may produce 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 following code shows the management task S used in Figure 2.1 to handle the data synchronisation:

```
for (i=0; i<(M-1); i++) {
    a[i*3] = tmp0[i];
    a[i*3+1] = tmp1[i];
    a[i*3+2] = tmp2[i];
}
tc = M * 3 - N - 1;
switch(tc) {
    case 0:
        a[(M-1)*3] = tmp0[M];
        a[(M-1)*3+1] = tmp1[M];
        a[(M-1)*3+2] = tmp2[M];
        break;
    case 1:
        a[(M-1)*3] = tmp0[M];
        a[(M-1)*3+1] = tmp1[M];
        break;
    case 2:
        a[(M-1)*3] = tmp0[M];
```

```
            break;
    }
```

where M is the actual count of executing the unrolled loop, N is the original loop count. $tmp0$, $tmp1$ and $tmp2$ are the results produced by unrolled iterations $B1$, $B2$ and $B3$ respectively. a is the original array to store results.

If there is data dependency between iterations, the functionality of the management task is to select the correct result from unrolled iterations. The following code shows the management task SB used in Figure 2 to select result from $B1$, $B2$ and $B3$:

```
    tc = M * 3 - N - 1;
    switch(tc) {
        case 0:
            a = tmp2;
            break;
        case 1:
            a = tmp1;
            break;
        case 2:
            a = tmp0;
            break;
    }
```

By generating these management tasks to handle data synchronisation, the generated mapping/scheduling solution is independent of designer's fault or run-time condition. E.g. For instance, if a user specifies a loop count during compile-time, the loop is unrolled based on this information, and a mapping/scheduling solution is generated. If the loop count match the run-time loop count, maximum performance can be achieve. However, if the loop count during run-time is different, the generated data management task can handle data synchronisation dynamically, which means the generated mapping/scheduling solution is still feasible.

3 Results

3.1 Experimental setup

The reference heterogeneous computing system used in this work contains one AMD Opteron(tm) Processor 2218 at 2.6GHz and one Celoxica RCHTX-XV4 FPGA board with a Xilinx Virtex-4 XC4VLX160 FPGA. Both the FPGA board and microprocessor are connected by HTX interface with maximum data transfer up to 3.2GB/s.

An isolated word recognition system [5] is used as an application which uses 12th order linear predictive coding coefficients (LPCCs), a codebook with 64 code vectors, and 20 hidden Markov models (HMMs), each with 12 states. One set of utterances from the TIMIT TI 46-word database [4] containing 5082 words from 8 males and 8 females are used for recognition. CPU profiling results show that loops in vector quantisation (VQ), autocorrelation (AUTOCC) and hidden Markov model decoding (HMMDEC) consume the largest CPU resource, which are 71.19%, 15.4% and 6.11% respectively.

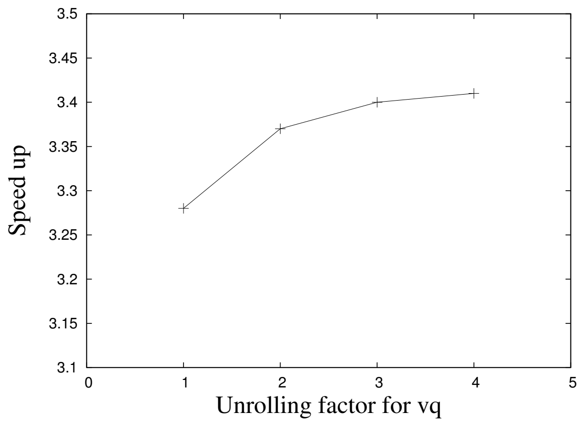


Figure 3. Speed up for different unrolling factors of vector quantisation.

3.2 Loop unrolling

To evaluate the quality of a mapping/scheduling solution, a speed up factor is used, it is calculated as the processing time using single CPU divided by the processing time using the heterogeneous computing system. Figure 3 shows the speed up for different unrolling factors of VQ, where all processes of the isolated word recognition system are executed on CPU except vector quantization. It is found that speed up increases with unrolling factor. However, it tends to flatten when the unrolling factor increases especially beyond 3, this configuration is thus used for the following experiment.

Unrolling vector quantisation for 3 iterations, Figure 4 shows the speed up for different unrolling factors of HMMDEC inner loop and AUTOCC. The best speed up is obtained when both HMMDEC and AUTOCC are unrolled for 12 iterations, AUTOCC is actually fully unrolled. The FPGA resource used is 48039 slices and the operating frequency is 319MHz. The speed up obtained for this configuration is 9.8, compared with a speed up of 4.7 obtained without unrolling, where VQ, AUTOCC, and HMMDEC are executed in FPGA without unrolling, a 2.1 times of enhancement is obtained using unrolling.

3.3 Compile-time vs run-time parameters

In the previous experiment, mapping/scheduling is generated based on a compile-time 12th order LPCCs. Figure 5

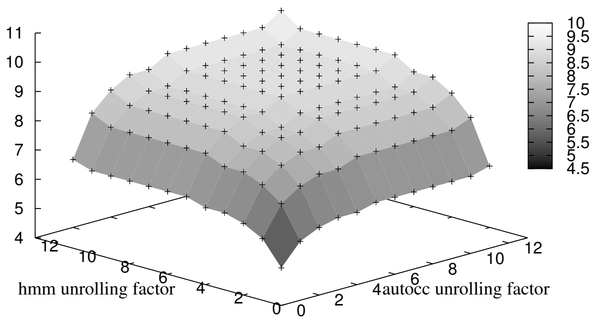


Figure 4. Speed up for different unrolling factors of hidden Markov model decoding and autocorrelation.

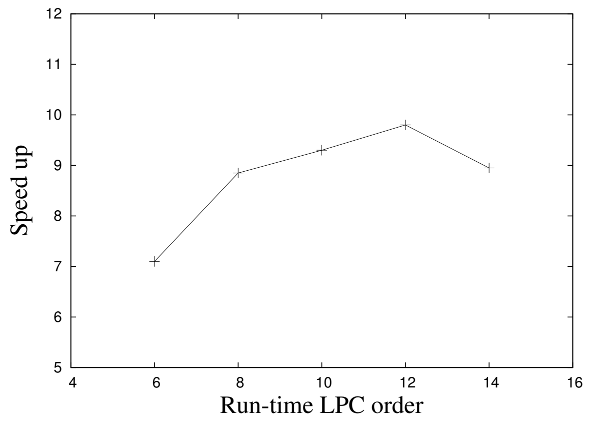


Figure 5. Speed up for different run-time LPCC order, the compile-time LPCC order is 12.

shows the performance of the system for different run-time LPCC orders. It is found that maximum performance is achieved on 12th LPCCs, and performance is drop when the run-time LPCC order is different from pre-defined value during compile-time, but the proposed strategy can still provide a feasible system.

4 Conclusions

A static task graph mapping/scheduling technique with loop unrolling is proposed and the generated mapping/scheduling is tolerant of designer's fault or run-time condition. Experimental results obtained using an isolated speech system show that a speed up of 9.8 times is achieved and it outperforms an approach without unrolling by 2.1 times. The performance drops gradually when compile-time and run-time parameters are different.

Acknowledgement. The support of FP6 hArtes (Holistic Approach to Reconfigurable Real Time Embedded Systems) Project, the UK Engineering and Physical Sciences Research Council, Agility, Celoxica and Xilinx is gratefully acknowledged.

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, P. H. W. Leong, and W. Luk. 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] LDC. *http://www.ldc.upenn.edu.*

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.