MULTI-LOOP PARALLELISATION USING UNROLLING AND FISSION

Yuet Ming Lam, José Gabriel F. Coutinho, Chun Hok Ho, Philip Heng Wai Leong, Wayne Luk

ABSTRACT

A technique for parallelising multiple loops in a heterogeneous computing system is presented. Loops are first unrolled and then broken up into multiple tasks which are mapped to reconfigurable hardware. A performance-driven optimisation is applied to find the best unrolling factor for each loop under hardware size constraints. The approach is demonstrated using three applications: speech recognition, image processing, and the N-Body problem. Experimental results show that a maximum speedup of 34 is achieved on a 274MHz FPGA for the N-Body over a 2.6GHz microprocessor, which is 4.1 times higher than an approach without unrolling.

1. INTRODUCTION

Microprocessors are commonly used to implement computing systems as they have the advantages of low cost and fast development time. In performance-critical applications, performance can be improved by introducing larger degrees of spatial parallelism via reconfigurable hardware implemented on field programmable gate arrays (FPGAs). Heterogeneous computing systems using both microprocessors and FPGA-based custom function units can combine advantages of both for many applications.

Computational intensive tasks in digital signal processing algorithms are usually iterative operations. Scheduling such loops in a heterogeneous computing system to fully utilise the available resources is difficult due to their complex nature. Techniques which have been previously proposed tend to address single loop only and are summarised as follows.

- Control flow based [1][2]. This approach divides a control flow graph into various sub-graphs based on control edges and each sub-graph is scheduled independently, typically list scheduling technique is used. A complete scheduling is generated by combining all the schedulings of sub-graphs. This approach only analyse one iteration of the loop body, it doesn't target on generating higher parallelism implementation for multiprocessor systems.

- Modulo scheduling [3]. Generates a schedule for one iteration of a loop so that all iterations repeat at a fixed interval, i.e. a software pipelined design. Since only a single iteration is analysed, limited parallelism is achieved.

- Graph conversion [4]. An application with loop can be characterised as a cyclic graph, this approach attempts to find a better scheduling of the loop body by using a graph traversal algorithm to convert the cyclic graph to an acyclic one with minimised critical path. Depth first search technique is used to traverse the cyclic graph and remove the feedback edge, an acyclic graph scheduling technique is then used to form a scheduling of the loop body. This approach does not analyze task dependency in different iterations which may result in reduced parallelism.

- Loop unrolling [5][6][7]. 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. These references have only been applied to parallelise a single loop.

- Dynamic scheduling [8]. This approach schedules tasks at run-time making use of both online and offline parameters. The loop condition is checked dynamically at runtime. Loop parallelisation is not addressed in this approach.

- Loop fission [9][10]: This approach breaks a loop into multiple tasks and maps each individual one to FPGA. Implementing applications which exceed the size constraint on FPGA thus becomes feasible. Since loop unrolling is not involved, this approach results in limited parallelism.

A comparison between this work and different approaches are shown in Table 1. Previous work has focused on parallelising a single loop [3],[4],[5],[6],[7] and multi-loop optimisation has not been adequately addressed. Since reconfigurable hardware in a heterogeneous system is capable of supporting parallel execution of tasks, a major challenge is to develop techniques which can effectively exploit this capability.

This work explores techniques to optimise applications with multiple loops in a heterogeneous computing system. Our recent work has shown that an integrated mapping and scheduling scheme with multiple neighborhood functions [11],

Table 1. Some approaches to address mapping/scheduling.

references	approach	examples of applications	comments
[1] [2]	Control flow based	GCD, counter, Filtering	multiprocessors system not addressed
[3]	Modulo scheduling	DCT, FFT	analyze one iteration, single loop
[4]	Graph conversion	random graphs	less parallelism, single loop
[5] [6] [7]	Loop unrolling	random graphs, FFT, solver equalizer	single loop unrolling
[8]	Dynamic scheduling	fractal generation	loop unrolling not addressed, single loop
[9][10]	Loop fission	JPEG compression, DCT, BPIC	loop unrolling not addressed, single loop
this work	Multi-loop unrolling	speech system image processing, N-Body	global unrolling factors determining, coarse-grained, heterogeneous systems

and combining mapping and scheduling with loop unrolling [12] can achieve considerable performance gains. This work complements those results through a method for optimising the unrolling factors in multiple loops. The novel aspects of this work are as follows:

- A performance-driven strategy, combined with an integrated mapping/scheduling system with multiple neighborhood functions, to find the best unrolling factor for each loop (Section 2.4).
- A static mapping and scheduling technique capable of handling cyclic task graphs for which the number of iterations are not known until run-time (Section 3.1 and Section 3.3).
- The introduction of additional management tasks for dynamic data synchronisation while maintaining near optimal performance when an accurate compile-time prediction of the run-time condition is made (Section 3.2).

The remainder of this paper is organised as follows. The proposed multi-loop parallelisation scheme is presented in Section 2. Section 3 introduces the loop unrolling technique and provides an overview of the multiple neighborhood function based mapping/scheduling system. Experimental results are given in Section 4, and finally, concluding remarks are given in Section 5.

2. MULTI-LOOP PARALLELISATION

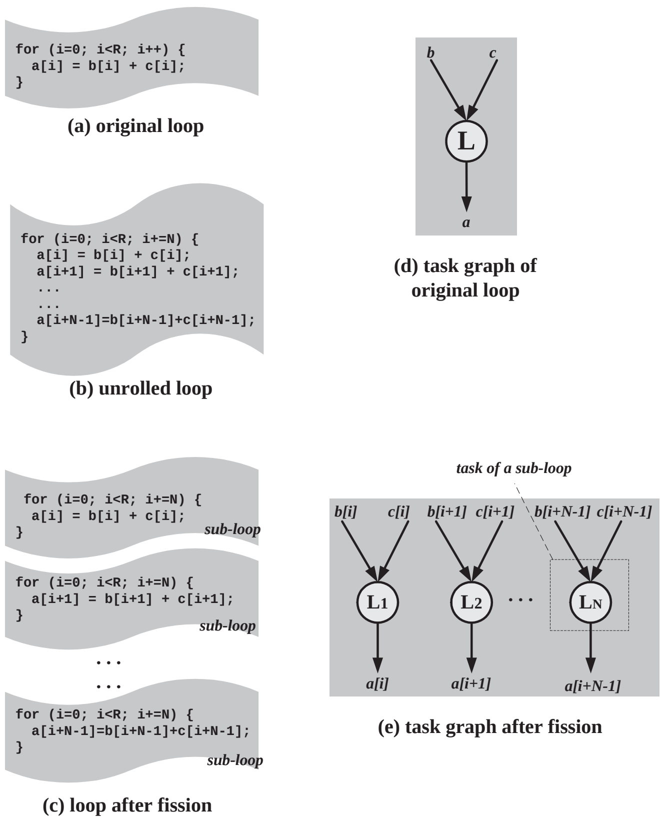
2.1. Reference architecture

The reference heterogeneous computing system contains two processing elements (PEs): one microprocessor and one FPGA. Each processing element has a local memory for data storage during task execution, and the communication channel between these two processing elements is being assigned a weight which specifies the data transfer rate. Results of a task's predecessors must be transferred to the local memory before this task starts execution.

2.2. Notations

Given an application containing a loop (Figure 1a), the following are various notations used in this paper:

- **Loop unrolling and unrolling factor:** Loop unrolling is a process to duplicate the body of a loop multiple times and use them to replace the original body, where the loop-control code is adjusted accordingly. The number of copies being duplicated is called unrolling factor. For example, Figure 1b shows an unrolled loop with an unrolling factor of N.
- **Loop fission and sub-loop:** Loop fission is a process to split a loop that contains multiple instructions into a number of loops with the same loop control. Each splitted loop is called a sub-loop which contains a portion of instructions of the original loop body. For instance, Figure 1c shows multiple sub-loops after fission.
- **Task:** A task is a block of consecutive instructions derived from task partitioning stage for a given application [13], e.g. a loop in Figure 1a is a task.
- **Task graph:** A task graph is an acyclic graph representing the data flow dependencies of tasks, where


```
for (i=0; i<R; i++) {
  a[i] = b[i] + c[i];
}
```

```
for (i=0; i<R; i+=N) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  ...
  ...
  a[i+N-1]=b[i+N-1]+c[i+N-1];
}
```

```
for (i=0; i<R; i+=N) {
  a[i] = b[i] + c[i];
}
for (i=0; i<R; i+=N) {
  a[i+1] = b[i+1] + c[i+1];
}
...
...
for (i=0; i<R; i+=N) {
  a[i+N-1]=b[i+N-1]+c[i+N-1];
}
```

Fig. 1. Examples showing various notations: (a) Loop. (b) Unrolled loop. (c) Obtained loop after fission. (d) Task graph representating the original loop, where the loop is a node L in the graph. (e) Task graph representing the loops after fission.

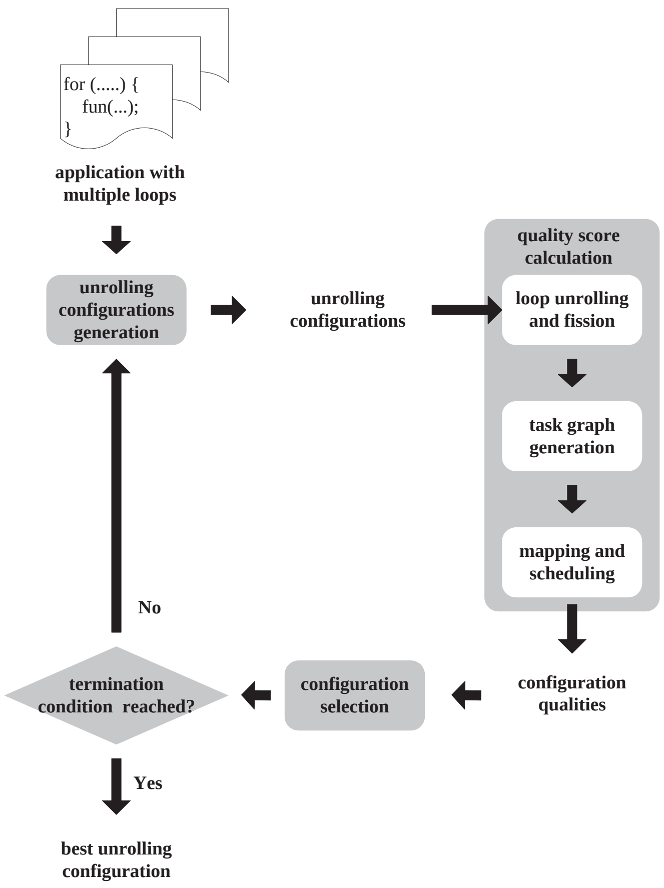

for (.....) {
fun(...);
}
application with multiple loops
unrolling configurations generation
unrolling configurations
quality score calculation
loop unrolling and fission
task graph generation
mapping and scheduling
configuration qualities
configuration selection
termination condition reached?
No
Yes
best unrolling configuration

Fig. 2. Overview of the proposed multi-loop parallelisation strategy.

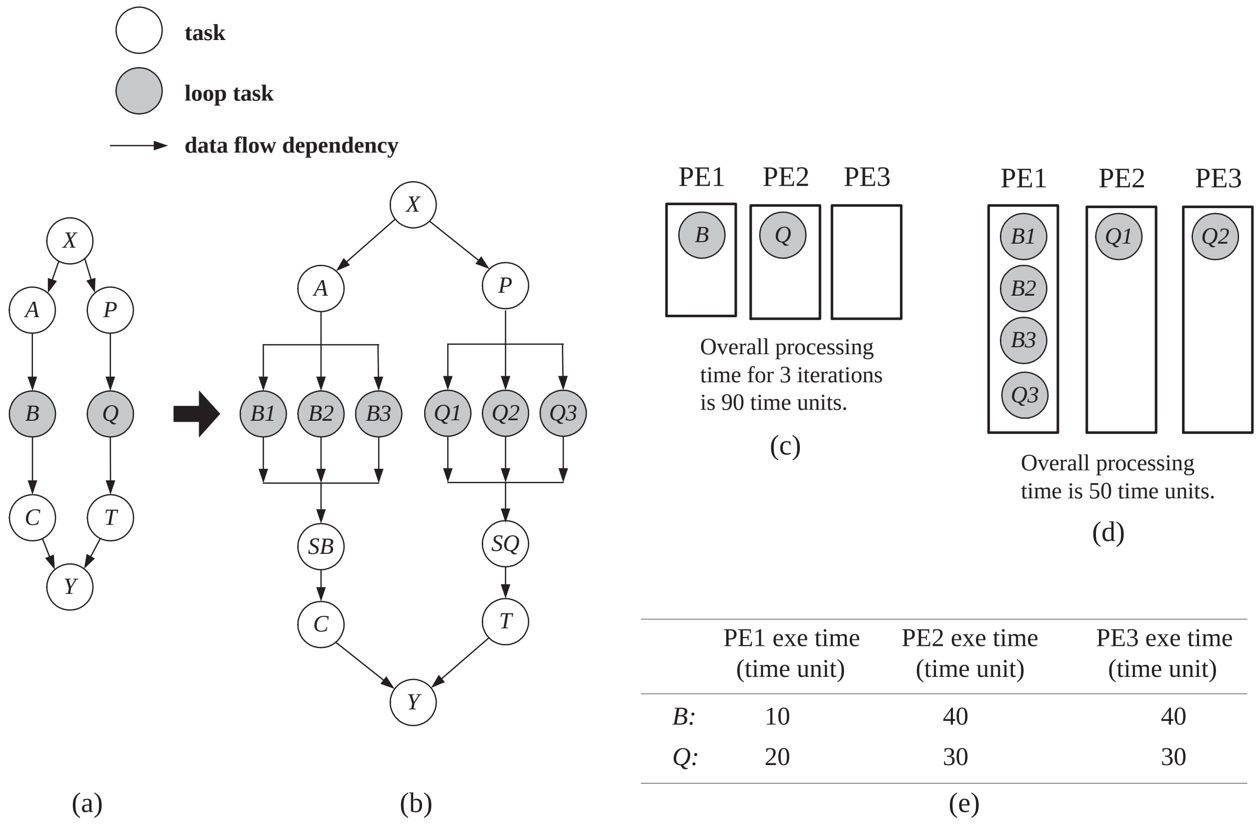
a task in the task graph is only executed once and it cannot be executed prior to its predecessors due to the data dependency. For instance, Figure 1d and Figure 1e are two task graphs of Figure 1a and Figure 1c respectively, where each loop is a node in the graph.

2.3. Overview

Figure 2 gives an overview of the proposed multi-loop parallelisation strategy. A search strategy is employed where the goal is to find an optimal unrolling factor for each loop so that the overall performance is maximised. This section focuses on the search of unrolling factors, the calculation of quality score will be introduced in Section 3.

Given an application containing a set of loops $LP = \{lp_1, lp_2, ..., lp_n\}$, let $UC = \{uc_1, uc_2, ..., uc_m\}$ be a set of unrolling configurations with each $uc_i = \{uf_1, uf_2, ..., uf_n\}$ designating an instance of the unrolling factors of all loops, where uf_j is the unrolling factor of loop j. Each unrolling configuration uc_i thus contains all unrolling factors for all loops in this application. In each iteration of the search, a set of unrolling configurations UC is firstly generated and a quality score is then calculated for each configuration after loop unrolling and fission, task graph generation, and mapping/scheduling processes have been applied. The best unrolling configuration uc_i with highest quality score is selected and used for the next iteration. This process is repeated iteratively until a termination condition is reached, the goal being to find a solution with the maximum quality score.

The advantage of considering unrolling and fission of all loops globally is that unrolled sub-loops from various loops can be potentially executed in parallel. This allows for a better mapping/scheduling solution to be found after unrolling and fission. Figure 3 shows an example of unrolling two loops which have no data dependencies between iterations. In the original graph, B and Q represent two loops, $B1$, $B2$ and $B3$ are the three unrolled sub-loops of B, and Q is unrolled as $Q1$, $Q2$ and $Q3$. Before unrolling and fission, B and Q are mapped to two processing elements PE1 and PE2.


task
loop task
data flow dependency
X
A
P
B
Q
C
T
Y
B1
B2
B3
Q1
Q2
Q3
SB
SQ
PE1
PE2
PE3
Overall processing time for 3 iterations is 90 time units.
Overall processing time is 50 time units.
(a)
(b)
(c)
(d)
(e)

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

Fig. 3. Mapping/scheduling example for loop unrolling and fission without data dependencies between iterations: (a) Task graph containing two parallel loops. (b) Task graph after unrolling the two loops for three iterations. (c) Mapping and scheduling solution before unrolling and fission, overall processing time for 3 iterations is 90 time units. (d) Mapping and scheduling solution after unrolling and fission, overall processing time is 50 time units. (e) Execution time of one iteration of the loop body for different processing elements. Higher inter-loop and intra-loop parallelism is achieved by unrolling two loops.

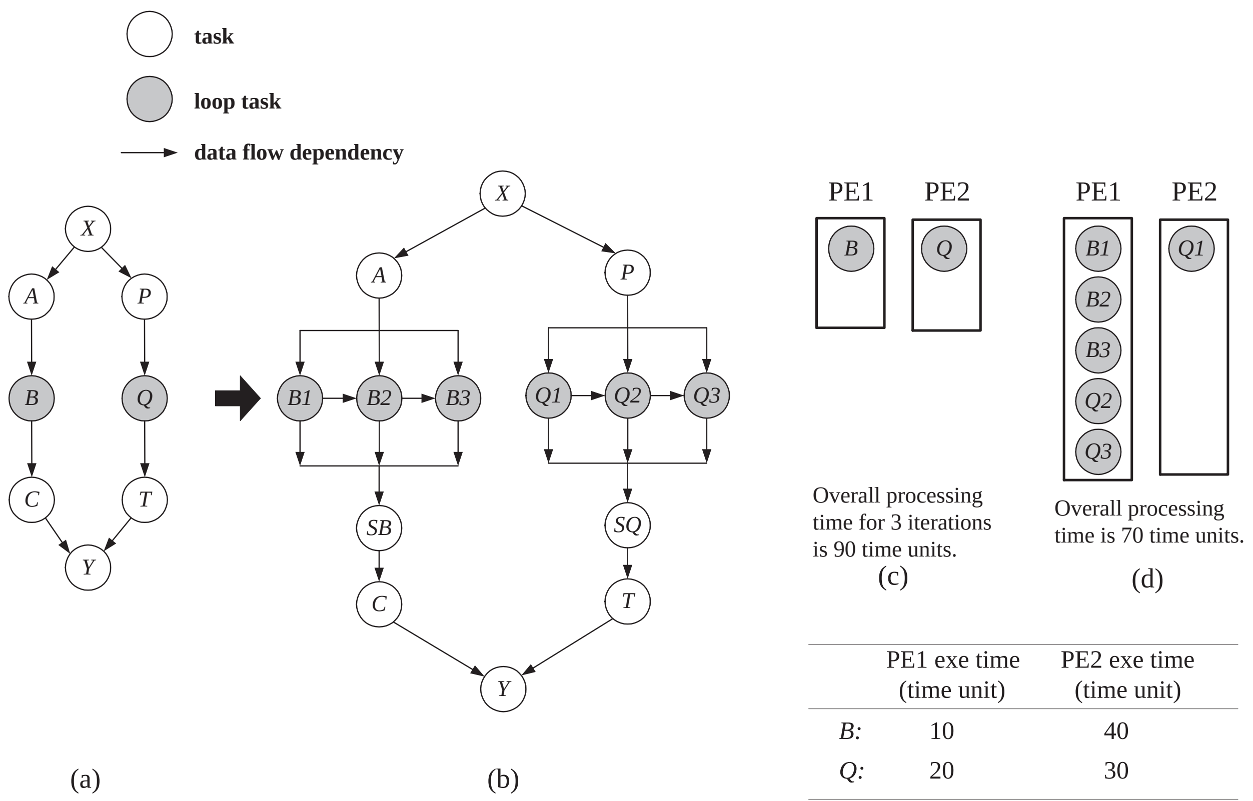
Hardware resources are not fully utilised and the processing time for three iterations using this mapping is 90 time units (Figure 3c). After unrolling and fission, the first two sub-loops ($Q1$ and $Q2$) are mapped to PE2 and PE3 respectively, and other unrolled sub-loops are mapped to PE1. Processing time is reduced to 50 time units (Figure 3d). Tasks SB and SQ are two generated management tasks to synchronise results produced by different sub-loops which will be introduced in Section 3.2.

Unrolling and fission can still achieve higher parallelism for loops with data dependency between iterations. As a loop may be executed in parallel with other tasks in an application, after unrolling and fission, execution sequence of unrolled sub-loops can be better combined with other tasks. Figure 4 shows the unrolling/fission of two loops with data dependencies between iterations. Before unrolling and fission, B is mapped to PE1 and Q is mapped to PE2. The overall processing time for three iterations is 90 time units (Figure 4c). After unrolling, the first sub-loop of Q (i.e. $Q1$) is mapped to PE2, and the remaining sub-loops ($Q2$ and $Q3$) can be executed in PE1. The overall processing time becomes 70 time units (Figure 4d). A better mapping/scheduling solution with higher inter-loop parallelism is thus obtained.

2.4. Generation and selection of unrolling configuration

If an application contains only one loop it obviously should be selected for unrolling. For the multiple loop case, the number of loops to unroll and the corresponding unrolling factors need to be determined. Since unrolling a loop without data dependencies between iterations is likely to achieve more performance gain than unrolling a loop with data dependencies, a performance-driven strategy (Algorithm 1) is proposed in this work.

Given an application containing a set of loops $LP =$


task
loop task
data flow dependency
X
A
P
B
Q
C
T
Y
B1
B2
B3
Q1
Q2
Q3
SB
SQ
PE1
PE2
Overall processing time for 3 iterations is 90 time units.
Overall processing time is 70 time units.
(a)
(b)
(c)
(d)

	PE1 exe time (time unit)	PE2 exe time (time unit)
B:	10	40
Q:	20	30

(e)

Fig. 4. Mapping/scheduling example for loop unrolling and fission with data dependencies between iterations: (a) Task graph containing two parallel loops having data dependency between iterations. (b) Task graph after unrolling the two loops for three iterations. (c) Mapping and scheduling solution before unrolling and fission, overall processing time for 3 iterations is 90 time units. (d) Mapping and scheduling solution after unrolling and fission, overall processing time is 70 time units. (e) Execution time of one iteration of the loop body for different processing elements. Higher inter-loop parallelism is achieved by unrolling two loops.

$\{lp_1, lp_2, ..., lp_n\}$, an initial unrolling configuration uc_{best} is generated with all unrolling factors uf_i being set to 1, i.e. $uc = \{uf_i\}$, where $uf_i = 1$ for $1 \leq i \leq n$. A new set of unrolling configurations $new\_uc = \{uc_1, uc_2, ..., uc_n\}$ is generated by incrementing each uf_i in turn, e.g. $uc_1 = \{2, 1, ..., 1\}$ and $uc_2 = \{1, 2, ..., 1\}$. For each unrolling configuration uc_i, a quality score qs_i is calculated by first applying the unrolling factors specified in uc_i followed by fission to break the new loop into sub-loops over the same loop count with each loop having the same loop body as the original loop. A task graph is then generated with each sub-loop being treated as a task. The task graph is then passed to the mapping and scheduling process, where a complete mapping/scheduling solution is generated and a quality score is calculated (Section 3). As a result, a set of quality scores $QS = \{qs_1, qs_2, ..., qs_n\}$ is produced. Afterward the corresponding quality improvement qi_i is calculated as:

$$qi_i = qs_i - qs_{best} \tag{1}$$

where qs_{best} is the best quality score to date. The best unrolling configuration uc_i with highest quality improvement qi_i is chosen, the best quality score qs_{best} is updated as qs_i and the unrolling configuration uc_{best} is replaced by uc_i. This process is repeated until the resources on the FPGA are exhausted, causing termination of the algorithm.

3. QUALITY SCORE CALCULATION

3.1. Unrolling, fission, and task graph generation

Given a set of loops $LP = \{lp_1, lp_2, ..., lp_n\}$ and an unrolling configuration $uc = \{uf_1, uf_2, ..., uf_n\}$. The following steps are used to generate a task graph:

Algorithm 1 Search the best unrolling configuration

```
1: $used\_fpga\_area \Leftarrow 0$
2: $uc_{best} = \{uf_i\}$, where $uf_i = 1$ for $1 \leq i \leq n$
3: $qs_{best} \Leftarrow 0$
4: **while** $used\_fpga\_area < total\_fpga\_area$ **do**
5:    **for all** loops $lp_i$ **do**
6:       $uc_i[uf_i] = uc[uf_i] + 1$
7:    **end for**
8:    **for all** unrolling configurations $uc_i$ **do**
9:       **for all** loops $lp_i$ **do**
10:          unroll $lp_i$ for $uf_i$ iterations, where $uf_i \in uc_i$
11:          loop fission
12:       **end for**
13:       generate new task graph
14:       generate complete mapping/scheduling $ms_i$
15:       calculate quality score $qs_i$ for $ms_i$
16:       $qi_i \Leftarrow qs_i - qs_{best}$
17:    **end for**
18:    find loop $i$ with maximum $qi$
19:    $qi_{best} \Leftarrow qi_i$
20:    $qs_{best} \Leftarrow qs_i$
21:    $ms_{best} \Leftarrow ms_i$
22:    $uc_{best} \Leftarrow uc_i$
23:    update $used\_fpga\_area$
24: **end while**
25: **return** $uc_{best}$ and $ms_{best}$
```

- Unroll each loop lp_i according to uf_i.
- Break each unrolled loop lp_i into uf_i sub-loops by fission, each sub-loop performs the same operations as the original loop body before unrolling.
- Construct a new task graph by treating each sub-loop as a task, each having the same parent and child tasks as the original task before unrolling.
- Generate a management task to synchronise results produced by different sub-loops (Section 3.2), and insert this task to the tails of all unrolled sub-loops in the task graph (tasks SB and SQ in Figure 3b), i.e. predecessors of the management task are the unrolled sub-loops, successors of the management task are the successors of the original loop.

The produced task graph is then presented to the mapping and scheduling tool to generate a quality score (Section 3.3), which guides the search.

3.2. 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 reorganised in the correct sequence (Figure 5). Another problem is loop count uncertainty, e.g. a loop may be unrolled n times but the actual loop count at run-time may not be a multiple of n. In this case some results must be discarded. To handle these problems, a management task which: collects data from different unrolled tasks; keeps track of the actual loop count at run-time; organises the collected data into the correct sequence; and discard unneeded data is introduced. The management task is treated as a normal task, inserted into the task graph and presented to the mapping/scheduling tool. For loops without data dependencies, the following pseudocode shows the data synchronisation process:

```
for (i = 0; i < (M-1); i++) {
    for (j = 0; j < N; j++) {
        rst[i*N+j] = d[j][i];
    }
}
tc = R - (M-1) * N;
for (i = 0; i < tc; i++) {
    rst[(M-1)*N+i] = d[i][M-1];
}
```

where M is the actual count of the unrolled loop being executed, R is the required loop count for the loop before unrolling, and N is the number of iterations being unrolled. d is the result produced by different unrolled iterations, e.g. $d[0]$ is the result produced by the first iteration. rst is the original array to store results. The second loop is used to collect the results of the last iteration and discard unneeded data, where tc is the number of data remaining.

If there are data dependencies between iterations, the management task must select the correct result from the unrolled iterations:

```
tc = M * N - R;
switch(tc) {
    case 0:
        rst = d[N-1];
        break;
    case 1:
        rst = d[N-2];
        break;
    ...
    ...
    case N-1:
        rst = d[0];
        break;
}
```

The generated mapping/scheduling solution does not require the designer to know the exact loop termination conditions using these management tasks. However, users can specify an estimated loop count at compile time. Loops are 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

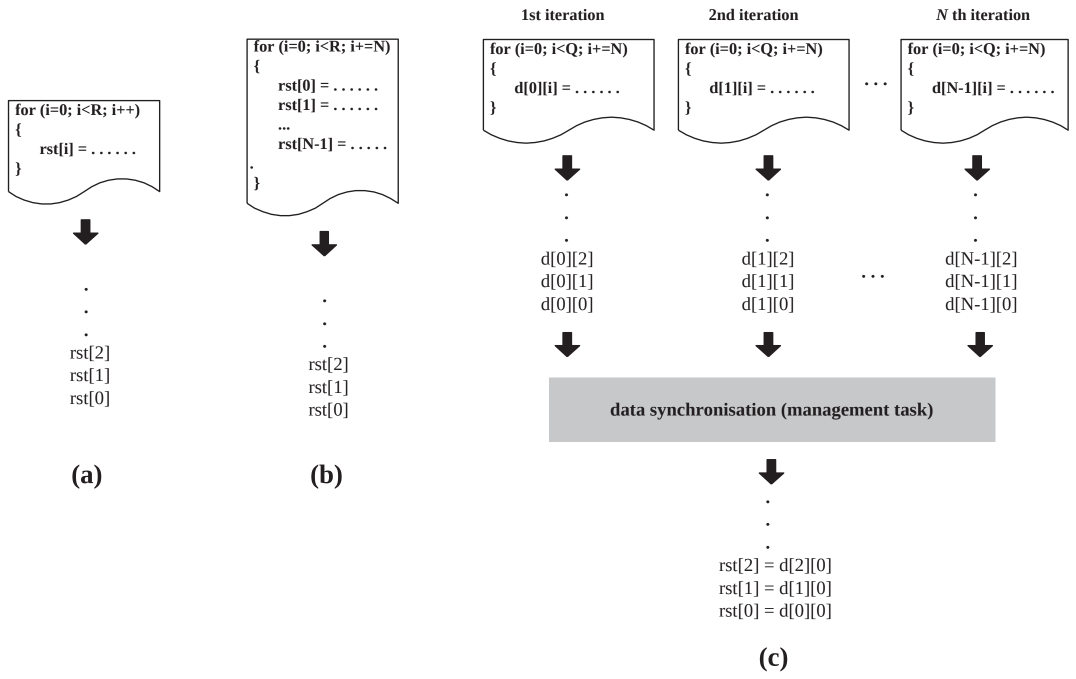


Fig. 5. Data synchronisation after unrolling and fission. (a) Original loop. (b) Unrolled loop. (c) Generated sub-loops after fission.

achieved. However, if the loop count is different, the 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.

3.3. Mapping and scheduling overview

A heuristic search-based approach is used to find the best mapping/scheduling solution for an input task graph as shown in Figure 6. Given a task graph and a target architecture specification which includes information concerning the processing elements and communications channel, a tabu search is used to iteratively generate different mapping/scheduling solutions (neighbors). For each solution, a speedup coefficient is calculated and used to guide the search with the goal being to find a solution with maximum speedup.

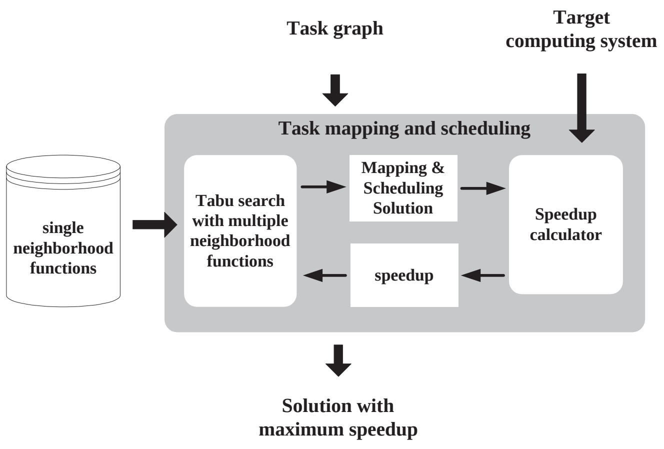

Task graph
Target computing system
Task mapping and scheduling
single neighborhood functions
Tabu search with multiple neighborhood functions
Mapping & Scheduling Solution
Speedup calculator
speedup
Solution with maximum speedup

Fig. 6. Overview of the mapping and scheduling system.

3.4. Integrated scheduling technique

Given a set of tasks $TK = \{tk_1, tk_2, ..., tk_n\}$ and a set of task lists $PL = \{pl_1, pl_2, ..., pl_m\}$, where each task list $pl_j = (as_{j1}, as_{j2}, ..., as_{jq})$ is an ordered task sequence to be executed by processing element pe_j. Each task in pl_j will be processed by pe_j in sequence when it is ready for execution, i.e. when all of its predecessors are finished. Task mapping and scheduling is 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 $pl_r \in PL$.

3.5. Multiple neighborhood functions

Tabu search is used to find the best mapping/scheduling solution. It is based on 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 the proposed tabu search technique with multiple neighborhood functions, after an initial solution is generated, two neighborhood functions are designed to move tasks between task lists and used to generate various neighbors simultaneously [11]. If there exists a neighbor better 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.

3.6. Quality score

For each mapping/scheduling solution, an overall execution time calculated, which is the time to process 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 a predecessor is assumed to be zero if they are assigned in the same processing element.

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

$$speedup = \frac{processing\ time_{single\ CPU}}{processing\ time_{Reference\ system}} \tag{2}$$

A higher speedup means a mapping/scheduling solution is better as the application can be finished using less time. This score is used to guide the tabu search and the goal is finding a solution with maximum speedup. This maximum speedup is used as the final output and defined as the quality score to measure the quality of the input unrolling configuration.

Table 2. Profiling results for major processes of the isolated word recognition system.

Process	% of exe time
vq	71.19
autocc	15.4
hmmdec	6.11
windowing	4.39
lpc_analysis	0.95
lpc2cep	0.93
find_max	0.39
others	0.64

Table 3. FPGA resources of different speech process, the total area is calculated by counting two “hmmdec12”.

Process	Area (slice)
vq3	21819
autocc12	10272
hmmdec12	15948
Total	63987

4. RESULTS

4.1. Experimental setup

The reference heterogeneous computing system used in work has one 2.6GHz AMD Opteron(tm) Processor 2218 and one Celoxica RCHTX-XV4 FPGA board with a Xilinx Virtex-4 XC4VLX160 FPGA. The FPGA board and microprocessor are connected via a HTX interface with maximum data transfer rate of 3.2GB/s.

An isolated word recognition (IWR) system [14] is used as an application. It 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 [15] containing 5082 words from 8 males and 8 females are used for recognition. Table 2 shows the profiling results of major processes of the isolated word recognition system on the AMD processor. It is found that loops in vector quantisation (vq), autocorrelation (autocc) and hidden Markov model decoding (hmmdec) consumed the most CPU resource, which are 71.19%, 15.4% and 6.11% respectively.

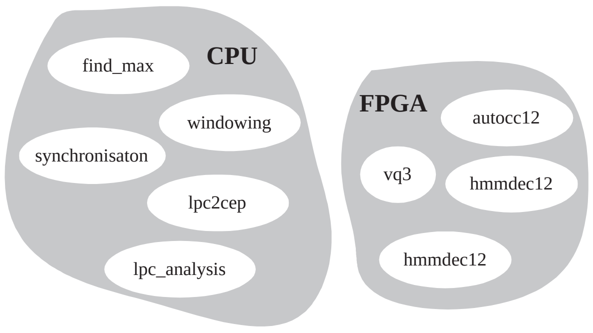


Fig. 7. Mapping the speech recognition system to the heterogeneous computing system. “vq3” are the unrolled 3 iterations of vector quantisation. “autocc12” are the unrolled 12 iterations for autocorrelation. “hmmdec12” are the unrolled 12 iterations for HMM decoding. Two “hmmdec12” means the outer loop of HMM decoding is unrolled for 2 iterations, i.e. decoding of two words are executed in parallel.

4.2. Multi-loop unrolling and fission

In this experiment, the proposed unrolling strategy is applied. Figure 7 shows the mapping of different processes in the speech system. It is found that vector quantisation is unrolled 3 times (vq3) and mapped to the FPGA, all 12 iterations of the autocorrelation process are unrolled (autocc12), inner loop of hidden Markov model decoding is unrolled for 12 iterations (hmmdec12) which is equal to the number of HMM states, the outer loop is further unrolled for 2 iterations which means two HMM decoding are executed in parallel. The corresponding FPGA resource usage is shown in Table 3 and the operating frequency is 318.7MHz, a speedup (quality score) of 10 is obtained for this configuration. In contrast, the speedup obtained without unrolling is 4.7, where vector quantisation, autocorrelation and HMM decoding are executed in FPGA without unrolling. An improvement of 2.1 times is hence obtained using the proposed strategy.

Figure 8 shows the speedups for different vector quantisation unrolling factors, where all other processes are executed on the CPU. It is found that the speedup increases with unrolling factor and saturates. This figure explains why only three iterations of vector quantisation are unrolled in the final mapping/scheduling solution.

4.3. Run-time vs compile-time parameters

In the above experiment, mapping/scheduling solutions are generated by assuming the LPCC order is 12 at compile-time. However, this value may be modified to cope with different circumstances at run-time. Using a mapping/scheduling solution generated with 12 LPCCs, Figure 9 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 LPCC order is different from compile-time value, e.g. a 5% drop at 10 LPCCs.

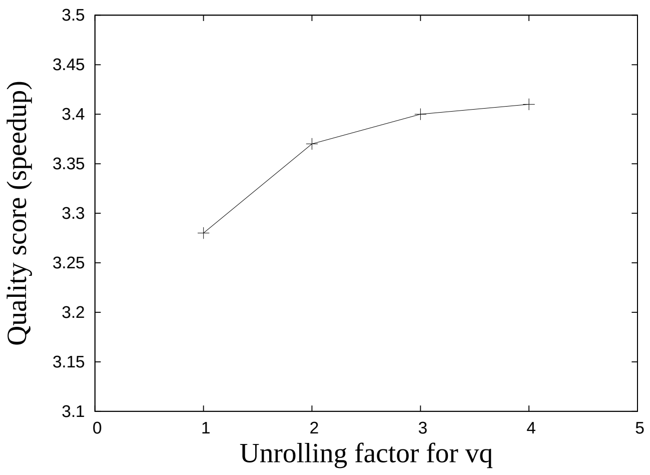


Fig. 8. Quality scores (speedups) for different unrolling factors of vector quantisation.

Table 4. FPGA resources and operating frequency for IWR, SUSAN, and N-Body.

Application	Area (slice)	Frequency (MHz)
IWR	63987 (94.7%)	318.7
SUSAN	60106 (88.9%)	274
N-Body	62139 (91.9%)	274

4.4. Quality score comparison

In addition to the IWR example, two other applications are employed to evaluate the proposed approach: the SUSAN corner detection image processing algorithm [16], and the N-Body problem [17]. Figure 10 shows the quality score comparison between strategies with and without unrolling. The FPGA resource usage and operating frequency are shown in Table 4. The proposed strategy can achieve 10, 19.7, and 34.3 times speedup for IWR, SUSAN and N-Body respectively, the corresponding improvements are factors of 2.1, 3.9, and 4.1 over the approach without unrolling. The improvements for SUSAN and N-Body are much higher than the 2.1 times improvement obtained using the IWR application because that there is a critical loop in each of these two applications: in SUSAN, the loop to compute the similarity of pixels and for N-body, the loop to compute velocity. Unrolling these loops significantly improves the performance

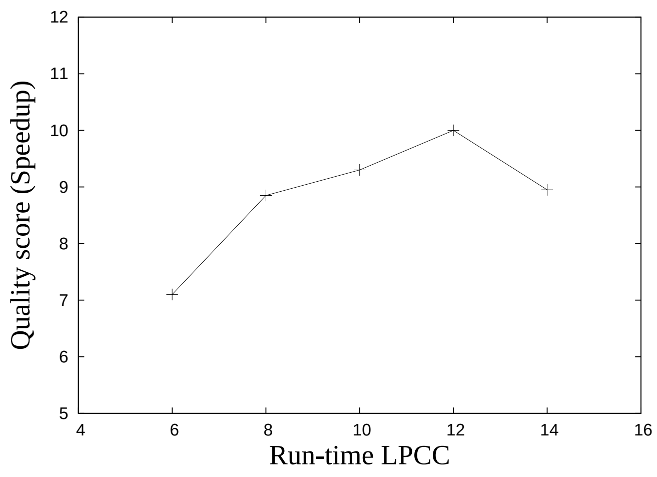


Fig. 9. Quality scores (speedups) for different run-time LPCCs, the estimated LPCC order during compile-time is 12.

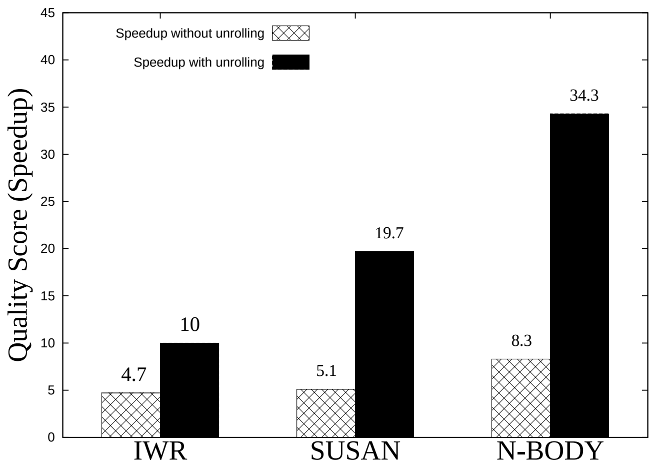


Fig. 10. Quality score (speedup) comparison between unrolling and without unrolling for different applications.

of those cases.

5. CONCLUSIONS

A multi-loop parallelisation technique involving fission and unrolling is proposed to improve intra-loop and inter-loop parallelism in heterogeneous computing systems. The utility of this approach is demonstrated in three practical applications and a maximum speedup of 34.3 times is obtained using a computing system containing an FPGA and a microprocessor. It is 4.1 times higher than the case where unrolling is not applied. The generated system is tolerant to run-time conditions, and its performance is closer to optimum when there is a more accurate prediction of run-time condition during compile-time.

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

6. REFERENCES

[1] R. Camposano, “Path-Based Scheduling for Synthesis,” *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 1, pp. 85–93, Janurary 1991.

[2] M. Rahmouni and A. A. Jerraya, “Formulation and Evaluation of Scheduling Techniques for Control Flow Graphs,” in *Proceedings of the Design Automation Conference*, 1995, pp. 386–391.

[3] 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*, 2007, pp. 1–8.

[4] F. E. Sandnes and O. Sinnen, “A New Strategy for Multiprocessor Scheduling of Cyclic Task Graphs,” *International Journal of High Performance Computing and Networking*, vol. 3, no. 1, pp. 62–71, 2005.

[5] T. Yang and C. Fu, “Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 6, pp. 608–622, June 1997.

[6] M. Weinhardt and W. Luk, “Pipeline Vectorization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 234–248, February 2001.

[7] 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*, 2006, pp. 1–10.

[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*, 2004, pp. 255–262.

[9] M. Kaul, R. Vemuri, S. Govindarajan, and I. Quaiss, “An Automated Temporal Partitioning and Loop Fission Approach for FPGA Based Reconfigurable Synthesis of DSP Applications,” in *Proceedings of the Design Automation Conference*, 1999, pp. 616–622.

[10] J. M. P. Cardoso, “Loop Dissevering: A Technique for Temporally Partitioning Loops in Dynamically Reconfigurable Computing Platforms,” in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003, pp. 22–26.

[11] 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*, 2008, pp. 275–280.

[12] ——, “Unrolling-based loop mapping and scheduling,” in *Proceedings of the International Conference on Field Programmable Technology*, 2008, pp. 321–324.

[13] W. Luk, J. G. F. Coutinho, T. Todman, Y. M. Lam, W. Osborne, K. W. Susanto, Q. Liu, and W. S. Wong, “A High-Level Compilation Toolchain for Heterogeneous Systems,” in *Proceedings of the International SOC Conference*, 2009, pp. 9–18.

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

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

[16] S. M. Smith and J. M. Brady, “SUSAN - A New Approach to Low Level Image Processing,” *International Journal of Computer Vision*, vol. 23, no. 1, pp. 45–78, 1997.

[17] S. J. Aarseth, “Direct Methods for N-Body Simulation,” *Multiple Time Scales*, 2001.