HIERARCHICAL LOOP PARTITIONING FOR RAPID GENERATION OF RUNTIME CONFIGURATIONS

Siew-Kei Lam[1], Yun Deng[1], Jian Hu[2], Xilong Zhou[2],
Thambipillai Srikanthan[1]

[1] Centre for High Performance Embedded Systems,
Nanyang Technological University,
50 Nanyang Avenue, Singapore
{assklam, dengyun, astsrikan}@ntu.edu.sg

[2] School of Software and Microelectronics,
Peking University, P.R. China

Abstract. Runtime reconfiguration provides an efficient means to reduce the hardware cost, while satisfying the performance, flexibility and power requirements of embedded systems. The growing complexity of the applications necessitates methods that can rapidly identify a suitable set of configurations by splitting the computational structures into temporal partitions in order to evaluate the benefits of runtime reconfiguration early in the design cycle. In this paper, we present a hierarchical loop partitioning strategy that reduces the complexity of the search space for determining the runtime custom instruction configurations for reconfigurable processors. Experimental results show that the proposed partitioning strategy can lead to an average and maximum performance gain (in terms of clock cycle savings) of over 14% and 31% respectively when compared to a recently reported technique. In addition, when compared to the existing technique, the proposed partitioning method has significantly lower runtime in many of the cases considered.

Keywords: FPGA, Runtime reconfiguration, temporal partitioning

1 Introduction

Emerging embedded applications for portable battery operated devices (e.g. mobile phones, PDAs, mobile gaming devices, etc.) necessitates computing platforms that are capable of meeting the increasing performance demands at low cost and low energy budget. At the same time, these computing platforms must maintain a high degree of flexibility to meet the shrinking time-to-market window. To this end, the instruction set extension capability of reconfigurable processors (e.g. [1]-[3]) provides an attractive means to meet these stringent requirements of embedded systems. A reconfigurable processor consists of a microprocessor core that is coupled with a RFU (Reconfigurable Functional Unit), which facilitates critical parts of the application to

be implemented in hardware e.g. FPGA (Field Programmable Gate Array) in the form of custom instructions.

Runtime reconfiguration offers the potential to realize cost efficient systems that can still lead to high performance by changing the configuration of a small reconfigurable hardware at runtime. The two main drawbacks that discourage the use of runtime reconfiguration in embedded real-time systems is the large reconfiguration overhead in commercial FPGA architectures, and the lack of supporting tools and methodologies.

In this paper, we present a framework that rapidly generates custom instruction configurations for a given application and evaluate the benefits of runtime reconfiguration on a reconfigurable processor with an area-constrained RFU.

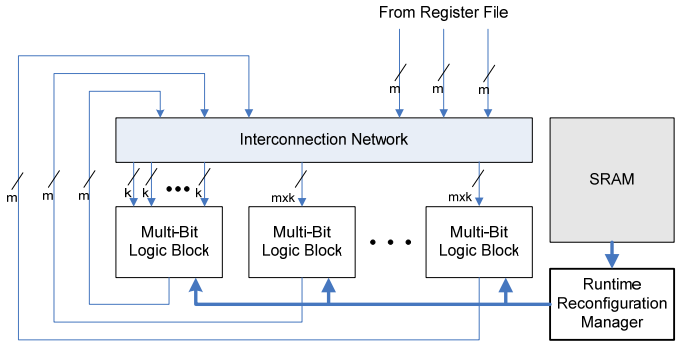

Figure 1: Target architecture

The target RFU model in Figure 1, which is described in detail in [4], consists of a set of multi-bit logic blocks that is organized around an interconnection network. Each multi-bit logic block incorporates programmable fine-grained logic elements that are similar to those found in commercial FPGA architectures. However, unlike commercial architectures, the logic elements within each multi-bit block shares the same configuration memory, which leads to reduce runtime reconfiguration overhead. In this paper, we assume that the full reconfiguration model is adopted, i.e. each reconfiguration will result in new configurations loaded onto all the logic blocks in the RFU. If the computation resource requirement of the custom instructions exceeds the number of available logic blocks in the RFU, then the custom instructions are mapped to different configurations. At runtime, a reconfiguration manager automatically loads the required configurations onto the logic blocks for computing the custom instructions. The proposed runtime management scheme relies on the dynamic execution profile to replace the functionality of the logic blocks with the goal of minimizing the overall reconfiguration overhead (see [4]).

1.1 Related Work

The potential benefits of instruction set extension have led to numerous amount of research work that focuses on generating custom instructions from a given application (see [5] for a list of references). In order to select custom instructions for different runtime configurations, temporal partitioning must be performed to divide the design into mutually exclusive configurations such that the computational resource requirement of each configuration is less than or equal to the reconfigurable resource capacity of the RFU.

The task partitioning algorithm presented in [6] for minimizing the communication cost is achieved in two steps. In the first step, an initial partition is obtained by using a network flow based algorithm to produce a set of feasible mean cuts. In the second step, a scheduling technique is employed to select an optimal global solution.

The work in [7] employs ILP (Integer Linear Programming) to achieve near-optimal latency designs for temporal partitioning of the application task graph. A loop transformation strategy was used to maximize the throughput while minimizing the runtime reconfiguration overhead.

The framework presented in [8] automatically partitions loops to a target platform consisting of a processor, RFU and memory hierarchy. A hierarchical loop clustering strategy is used to partition a loop into smaller clusters in order to perform optimal hardware-software partitioning of the loop clusters. The loop clustering strategy traverses the hierarchical loop graph in a top-down fashion and recursively clusters the nested loops until the sizes of all the clusters are within a pre-defined limit. The current framework in [8] does not allow multiple loops in a single configuration.

The work in [9] describes an architecture-aware temporal partitioning strategy for mapping custom instructions on an adaptive extensive processor, which incorporates coarse-grained functional units. The strategy partitions and modifies custom instructions that violate the RFU constraints, in order to map them onto the RFU.

Recently, a framework was presented in [10] to select custom instruction versions to be mapped onto appropriate configurations. A custom instruction version consists of a set of custom instructions from a particular loop that satisfy the area constraint of the RFU. The partitioning scheme consists of temporal partitioning of frequently executed application loops with custom instructions into one or more configurations, and spatial partitioning to select an appropriate custom instruction version for each loop within a configuration. The temporal partitioning problem has been modeled as a k-way graph partitioning problem, and spatial partitioning is resolved using dynamic programming. The framework assumes the availability of the custom instruction versions and their corresponding hardware area-time measures. This necessitates time-consuming hardware implementation of the custom instructions prior to the partitioning process if no high level estimation strategy is in place. In the worst case, the temporal partitioning algorithm in [10] iterates l times, where l is the number of hot loops.

1.2 Our Work

This paper presents a framework that rapidly partitions loops, which constitute the most frequently executed segments of embedded applications, into configurations and selects profitable custom instructions in the respective configurations. This enables the benefits of runtime reconfiguration to be evaluated early in the design cycle without undergoing time consuming hardware implementation. Unlike methods in [8] and [10], the proposed strategy takes into consideration the nested loop paths, and is not restricted to only hot loops which will vary with the input data. This can potentially lead to higher performance gain by leveraging on the target architecture's capability to perform dynamic execution profiling for determining suitable configurations to be loaded onto the RFU at runtime. Unlike the framework in [10], our work does not generate custom instruction versions and their corresponding hardware area-time measures prior to the partitioning process. Instead, we employ an efficient strategy that rapidly estimates the hardware area-time information of the custom instructions. In addition, the proposed framework relies on a hierarchical loop partitioning strategy that is similar to [8] for rapid partitioning of the application loops with custom instructions into one or more configurations. The proposed strategy utilizes efficient heuristics that takes into account the reconfiguration cost for partitioning loops into configurations. Finally, custom instructions for the respective configurations are then selected using a greedy algorithm. It is worth mentioning that the proposed method can be adopted in commercial FPGA tools as the target RFU model incorporates programmable logic elements that are similar to those found in commercial FPGA architectures.

2 Proposed Framework

Figure 2 gives an overview of the proposed framework. We have relied on the Trimaran compiler infrastructure [11] to generate the IR (Intermediate Representation) of the applications in the form of DFG (Data-Flow Graph). The IR serves as input to the Custom Instruction Generation stage to identify a set of custom instruction candidates. Details of the Custom Instruction Generation stage can be found in [5].

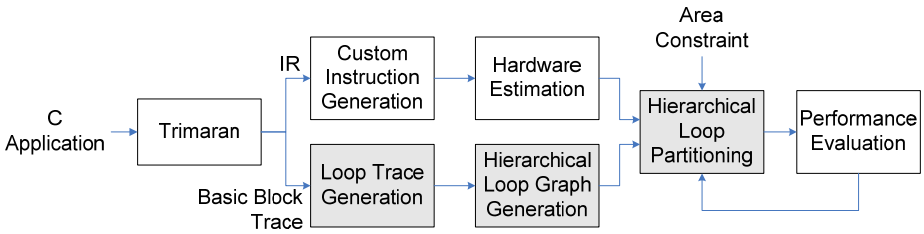

Figure 2: Proposed framework

The hardware area-time information of the custom instruction candidates are then rapidly estimated without undergoing time-consuming hardware implementation. This step estimates the area costs and critical path delays of the custom instruction candidates when they are implemented on the multi-bit logic blocks of the RFU. In this paper, we target programmable logic elements similar to those found in the Xilinx Virtex device [12]. It is noteworthy that the hardware area-time results using the proposed estimation technique have been shown to be within 8% of those obtained using hardware synthesis. In addition, the hardware estimation can be achieved in the order of milliseconds. The details of the hardware estimation process can be found in [5].

A hierarchical loop graph is then generated to enable rapid temporal partitioning of loops using the proposed hierarchical loop partitioning strategy. Note that the partitioning strategy relies on the hardware estimation results to obtain a profitable set of custom instruction configurations.

Finally, performance evaluation is performed using Trimaran's simulation environment, which converts the IR into executable codes and emulates the execution on a virtual HPL-PD processor [11].

3 Hierarchical Loop Generation

The proposed partitioning strategy relies on a HLG (Hierarchical Loop Graph) representation of the application in order to reduce the complexity of the search space for determining the runtime configurations. Figure 3 shows the HLG representation of a CFG (Control Flow Graph). Each node in the HLG (except for the root node *R*) represents a unique loop in the CFG. For example loop L_1 consists of the nested loop path with basic blocks 1, 2, 3, 5, 7 and 8. A directed edge between two nodes *s*, *d* in the HLG, where *s*, *d* are at different HLG levels (*s* is not the root node), signifies that *d* is a nested loop of loop *s*. The nodes in the HLG are also associated with a value f_x, which denotes the execution frequency of the corresponding loop.

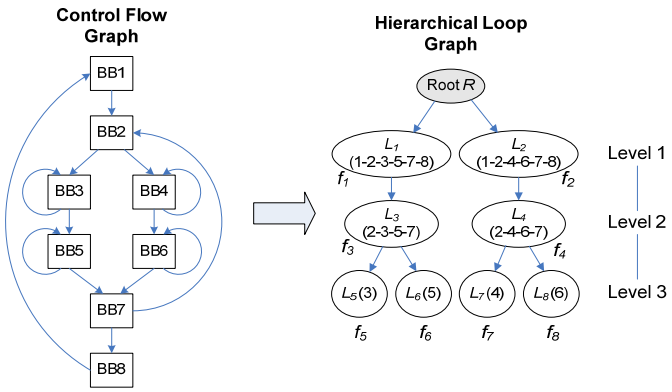

Figure 3: Hierarchical Loop Graph of CFG

Due to the difficulty in identifying loops across function boundaries in the Trimaran CFG, our current framework constructs the HLG from the application loop trace, which is derived from the basic block trace (see Figure 2). The basic block trace records the dynamic execution flow of the application basic blocks. The loop trace can be derived from the basic block trace by reading each basic block entry in the trace file and checking if it is part of a new loop or existing loop. A loop is identified when a particular segment of a trace consists of duplicated basic blocks. New or existing loops can be determined by maintaining a history of loops that have been identified so far. Finally, information pertaining to how the loops are nested within one another and the execution frequency of each unique loop are determined from the loop trace to construct the HLG.

4 Hierarchical Loop Partitioning Strategy

The proposed hierarchical loop partitioning consists of two main tasks: 1) temporally partition the application loops based on the HLG into one or more configurations, such that the overall performance gain of runtime reconfiguration is maximized, and 2) select profitable custom instructions from the loops in each configuration. The final output of the algorithm is a set of configurations and the selected custom instructions in each configuration.

4.1 Temporal Partitioning

Figure 4 illustrates the proposed temporal partitioning strategy, where L_x is a node in the HLG and C_y denotes a configuration. The partitioning algorithm aims to reduce the search space by giving preference to loops at the lower levels in the HLG and considering the higher levels (nested loops) only when they are necessary. Figure 4(a) shows an example of the Level 1 nodes in the HLG. The dummy root node, which is the parent of the Level 1 nodes, is included for programming consistency.

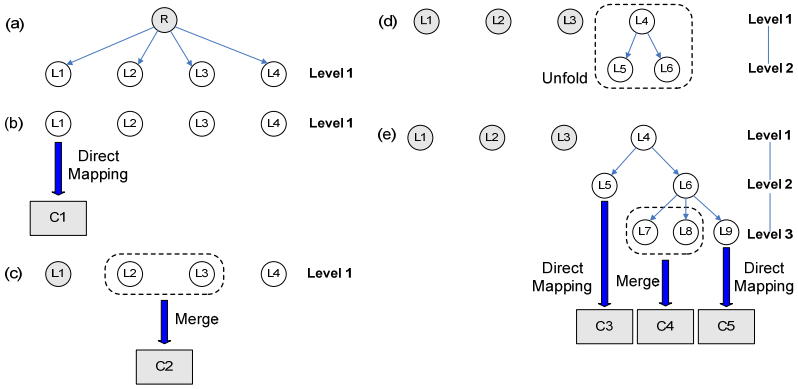

Figure 4: Example of proposed temporal partitioning strategy

The partitioning strategy evaluates the first loop in Level 1 of the HLG (i.e. L_1) and found that it can directly map onto a configuration (see Figure 4(b)). The loop can be directly mapped onto the configuration if the logic requirement of the custom instructions in that loop can efficiently utilize the resource capacity of the configuration. The partitioning algorithm then moves on to evaluate the next loop in Level 1 without the need to consider the nested loops of L_1. In the event that the custom instructions of a loop under-utilizes the resource capacity of the configuration, it is considered for merging with the next loop on the same level. This is shown in Figure 4(c) where L_2 and L_3 are merged into a single configuration. Note that the nested loops of L_2 and L_3 are not evaluated further. However, if the logic requirement of the custom instructions in the loop (e.g. L_4 in Figure 4(d)) is larger that the resource capacity of a configuration, an unfolding process may take place to allow the nested loops to be evaluated. It can be observed that only the immediate nested loops (or child nodes in the next higher level) are unfolded at a time. The evaluation process for direct mapping, merging and unfolding is then performed on these nested loops. For example in Figure 4(e), it can be observed that the nested loop L_5 can be directly mapped to a configuration, while the nested loop L_6 is further unfolded to the next higher level. In subsequent iterations, some of the nested loops of L_6 are merged into a configuration (i.e. L_7 and L_8), while others are directly mapped onto a configuration L_9). The process is repeated until all the loops in Level 1 of the HLG have been evaluated.

Figure 5 shows the algorithm for temporal partitioning. The temporal partitioning strategy consists of three main processes: 1) unfolding, 2) merging, and 3) direct mapping. Heuristics are employed to determine which of these three processes that the loop will be subjected to (i.e. line 3, 7 and 11). These heuristics are based on comparing the estimated area of all profitable custom instruction candidates in loop *x*, with a constant that is a product of the resource capacity of the configuration (i.e. *A*) and a pre-defined factor (i.e. **u** or **m**). The output of the temporal partitioning algorithm is a configuration set *C*, where each configuration in the set is associated with one or more loops in the HLG.

Let's first define a profitable custom instruction candidate *i* in loop *x* as one that satisfy the constraint in (1), where g_i^x is the gain of the profitable custom instruction candidate and t_{lb} is the reconfiguration time of a single multi-bit logic block, in terms of number of software clock cycles. a_i is the estimated area of *i* that is obtained using the method discussed in [5]. g_i^x is calculated as shown in (2), where f_x is the execution frequency of loop *x*, and n_i^x is the number of software clock cycles for *i.* We assume that each operation in the Trimaran IR takes one software execution clock cycle.

$$\frac{g_i^x}{a_i} > 2 \times t_{lb} \qquad (1)$$

$$g_i^x = f_x \times n_i^x \qquad (2)$$

```
Procedure Hierarchical_Loop_Partitioning (HLG, Ci)
Input:  Hierarchical Loop Graph (HLG),
        custom instruction candidates (Ci)
Result: Selected custom instructions for each
        configuration (Sel_CI)

1. configuration set C = empty;
2. Temporal_Partitioning (root);
3. Sel_CI = Cusom_Instruction_Selection (C, Ci);
4. return;

Procedure Temporal_Partiion (node)
Inputs: HLG, configuration set (C), configuration area (A),
        area of profitable custom instructions for each
        loop x in HLG (Ax)
Output: Set of configurations C

1.  for each loop x that is a child of node
2.      if Ax > u*A then
3.          Unfold_Loop (x);
4.      else if Ax < m*A then
5.          Put loop x into stack for merging;
6.      else if m*A < Ax < u*A then
7.          Direct_Map (x);
8.      end if
9.  end for
10. if stack is not empty then
11.     Merge_Loop (stack);
12. end if
13. Initialize new configuration c and insert node in c;
14. if Compute_Effective_Gain (c) >
        Compute_Effective_Gain (∀ci ∈ C with nested
        loops of node) then
15.     Replace ci with c in C;
16. end if
17. return;

Procedure Unfold_Loop (x)
18. if loop x has nested loops in HLG then
19.     Temporal_Partition (x);
20. else
21.     Direct_Map (x);
22. end if
23. return;

Procedure Direct_Map (x)
24. Initialize new configuration c and insert loop x in c;
25. if Compute_Effective_Gain (c) > 0 then
26.     Insert c as a new configuration in C
27. end if
28. return;

Procedure Merge_Loop (stack)
29. Initialize new configuration c with unutilized area Ac = A;
30. for each loop x in stack
31.     if Ac > Ax then
32.         Include loop x in configuration c;
33.         Ac = Ac – Ax;
34.     else
35.         if Compute_Effective_Gain (c) > 0 then
36.             Insert c as a new configuration in C;
37.         end if
38.         Initialize new configuration c with unutilized
            area Ac = A - Ax;
39.         Include loop x in new configuration c;
40.     end if
41. end for
42. if c != { } && Compute_Effective_Gain (c) > 0 then
43.     Insert c as a new configuration in C;
44. end if
45. return;
```

Figure 5: Algorithm for temporal partitioning

The constraint in (1) ensures that only custom instruction candidates that can still lead to notable performance gain after taking into account the runtime reconfiguration overhead are considered as profitable custom instructions. Line 2 in Figure 5 shows the heuristic used to evaluate if the current loop x needs to be unfolded to the next higher level in the HLG. The unfolding process can only take place when the following conditions are satisfied. Firstly, given a configuration area A, the estimated area of all the profitable custom instruction candidates ($A_x = \sum_{i}^{n\_pro} a_i$, where *n_pro* is the number of profitable custom instructions in loop x) must be larger than **u** x A, where **u** is a pre-defined unfolding factor ($1 \le u < 2$).

Secondly, in order for the loop to be unfolded, it must contain nested loops (see line 18). If these conditions are met, the unfolding process recursively calls the temporal partitioning algorithm, where the immediate nested loops of loop x will be unfolded (line 19). If the first condition is met but the second condition is violated, loop x is directly mapped onto a single configuration (line 21).

The heuristic that is used to consider if loop x should be merged with other loops in the same nested level of the HLG is shown in Line 4 of Figure 5. In particular, if the estimated area of all the profitable custom instruction candidates in loop *x* is less than **m** x *A*, where **m** is a pre-defined merge factor ($m < 1$), then the loop will be pushed into a stack to be considered for merging (Line 5). Note that these loops have been inserted into the stack as each of them will under utilize the logic capacity of the RFU.

When all the loops in a particular level have been considered, the loops that are inserted in the stack are partitioned into configurations. The merging process (line 29-44) employs a greedy approach to merge the loops in the stack until the total estimated area of the profitable custom instruction candidates in the merged loops exceed the given configuration area *A* (line 31-33). When the current configuration cannot accommodate a loop in the stack due to the area constraint, a new configuration is created for the loop (line 38-39). A configuration of merged loops is considered as a valid configuration only if the effective gain of the configuration is larger than 0 (line 35 and 42).

The effective gain of configuration *c* (G_c), that is calculated using the Compute_Effective_Gain function, is the total gain of all the profitable custom instruction candidates in the loops of *c* that have been greedily selected by taking into account the runtime reconfiguration overhead. G_c is calculated as shown in (3), where n_rtr_c is the number of times configuration *c* will be reconfigured in the application and *n_ml* is the number of loops that have been merged in configuration *c*. n_rtr_c can be determined from the loop trace. t_{config} is the overhead to reconfigure all the logic blocks (i.e. $t_{config} = n\_lb \times t_{lb}$, where *n_lb* is the number of logic blocks). The merging process terminates when all the loops in the stack has been considered for merging.

$$G_c = \sum_{x}^{n\_ml} \sum_{i} g_i^x - (t_{config} \times n\_rtr_c) \quad (3)$$

The heuristic in Line 6 of Figure 5 is used to determine if loop *x* can be directly-mapped to a single configuration. In particular, loop *x* can be directly mapped to a single configuration if the estimated area of all the profitable custom instruction candidates in loop *x* is 1) larger than **m** x *A*, and 2) less than **u** x *A*. It can be observed that similar to the merging process, a configuration is valid only if its effective gain is larger than 0 (line 25-26).

Note that the algorithm may eventually discard the configurations resulting from the unfolding process if it does not lead to higher performance gain (lines 14-15).

4.2 Custom Instruction Selection

In the temporal partitioning process, we have identified a set of configurations and their associated loops. The next step in the proposed partitioning strategy is to select custom instructions for each of the configurations. This is achieved by employing a greedy algorithm to select custom instructions with the highest gain in each configuration such that the total area required by the selected custom instruction does

not exceed the logic capacity of the RFU. The gain of the custom instructions in the loops associated with the configuration is first estimated using (2) and sorted in decreasing gain. The greedy algorithm then selects the custom instructions by giving preference to those with the highest gain in the sorted list such that the area of the selected custom instructions does not exceed the logic capacity. This process is repeated for all the configurations.

5 Experimental Results

In order to evaluate the benefits of the proposed partitioning strategy, we have employed six widely-used embedded benchmarks from [13]-[15]. We compare the proposed hierarchical partitioning strategy (denoted as *Hierarchical*) with a recently reported iterative method [10] (denoted as *Iterative*). Both methods perform temporal partitioning of the application loops and selection of custom instructions from the temporal partitions. For the Hierarchical method, we have empirically determined suitable values of **u** and **m** to be 1.2 and 0.6 respectively. We also assume that t_{lb} = 3K clock cycles (similar to the configuration time of one hardware unit in [10]). We have employed the full basic block trace for all the applications considered except for mpeg2 enc, where a partial basic block trace file is used to generate the loop trace and HLG as the original basic block trace file is very large. Similar to [10], we assume that the hardware area constraint is about 20-30% of the maximum hardware area that is required to accommodate all the custom instructions such that runtime reconfiguration is not necessary.

Table 1 compares the performance gain of the two methods. The performance gain is measured in terms of the clock cycle savings that resulted from instruction customization after taking into account the runtime reconfiguration cost. It can be observed that Hierarchical outperforms Iterative in all cases. This is due to the fact that Hierarchical takes into consideration the nested loop paths and is not restricted to hot loops. Hierarchical achieves an average and maximum performance gain of over 14% and 31% respectively when compared to Iterative.

Table 1: Comparison of performance gain

Clock Cycle Savings (K Cycles)			
	Iterative	**Hierarchical**	**%Gain**
Adpcm Enc	2030	2056	1.28
Cjpeg	1993	2310	15.93
Mpeg2 Enc	2917	3629	24.41
Virterbi00	173640	184695	6.37
Epic	998	1053	5.51
Sha	44309	58458	31.93

Table 2 compares the partitioning runtime between the two methods. For the proposed Hierarchical method, the runtime is measured for the tasks described in Section 4. Similarly, the runtime for the Iterative method is only measured for temporal and spatial partitioning. The time taken to generate the custom instruction versions and the corresponding hardware area-time measures in [10] is not considered. Both methods rely on the custom instruction generation process discussed in [5], and hence the time taken to identify the custom instructions is not considered in the comparison.

It can be observed that the runtime of Hierarchical is significantly lower than Iterative in most cases (i.e. Adpcm Enc, Cjpeg and Virterbi00), and comparable in the remaining cases (the difference in runtime in these remaining cases is less than 0.04s). This is due to the fact that the proposed strategy significantly reduces the search space of the loop partitioning process by evaluating the nested loops only when the profitable custom instructions in the corresponding larger loops cannot be mapped entirely onto the RFU area.

Table 2: Comparison of partitioning runtime

Partitioning Runtime (seconds)		
	Iterative	**Hierarchical**
Adpcm Enc	1.43	0.01
Cjpeg	6.61	0.05
Mpeg2 Enc	6.62	0.10
Virterbi00	0.29	0.32
Epic	0.27	0.27
Sha	0.04	0.05

6 Conclusion

We have presented a framework for reconfigurable processors that employs a hierarchical partitioning strategy which aims to maximize the performance of custom instruction realization through runtime reconfiguration, while minimizing the reconfiguration overhead. The proposed hierarchical partitioning strategy heuristically determines whether the loops in the HLG can be directly mapped to a configuration, merged with other loops or unfolded to the nested loops. Nested loops are only evaluated when the profitable custom instructions in larger loops cannot be mapped entirely onto a restricted RFU area. This strategy significantly reduces the search space of the partitioning process, resulting in rapid identification of temporal partitions. A greedy algorithm is then used to select custom instructions in each temporal partition. Experimental results show that the proposed hierarchical partitioning strategy can lead to a higher performance gain than a recently reported iterative partitioning approach. In addition, the partitioning runtime of the proposed partitioning strategy is significantly lesser than the iterative method in many of the cases considered. This enables rapid design exploration for maximizing the utilization

of reconfigurable space through runtime reconfiguration. Future work includes devising a method to generate the HLG from the application CFG and profiling information from Trimaran, as generating the HLG from the basic block trace is not feasible when the trace file is too large.

References

1. Altera: NIOS II Processors, http://www.altera.com/products/ip/processors/nios2/ni2-index.html
2. Xilinx Platform FPGAs, http://www.xilinx.com
3. Video/Imaging Design Line: Analysis: Stretch's Second-Gen Configurable Processor, http://www.videsignline.com/howto/videoprocessing/201311209 (2007)
4. Lam S.K. Huang F., Srikanthan T. and Wu J.: Run-Time Management of Custom Instructions on a Partially Reconfigurable Architecture, IEEE International Conference on Electronic Design (2008)
5. Lam S.K and Srikanthan T.: Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing, Journal of Systems Architecture, Vol. 55, No. 1, pp. 1-14 (2009)
6. Jiang Y.C. and Wang J.F.: Temporal Partitioning Data Flow Graphs for Dynamically Reconfigurable Computing, IEEE Transactions on Very Very Large Scale Systems, Vol. 15, No. 12, pp. 1351-1361 (2007)
7. Kaul M., Vemuri R., Govindarajan S. and Ouaiss I.: An Automated Temporal Partitioning and Loop Fission Approach for FPGA based Reconfigurable Synthesis of DSP Applications, Design Automation Conference, pp. 616-622 (1999)
8. Li Y., Callahan T., Darnell E., Harr O., Kurkure U. and Stockwood J.: Hardware-Software Co-Design of Embedded Reconfigurable Architectures, Design Automation Conference, pp. 507-512 (2000)
9. Mehdipour F., Noori H., Zamani M.S., Murakami K., Sedighi M. and Inoue K.: An Integrated Temporal Partitioning and Mapping Framework for Handling Custom Instructions on a Reconfigurable Functional Unit, Asia-Pacific Computer Systems Architecture Conference, pp. 219-230 (2006)
10. Huynh H.P., Sim J.E. and Mitra T.: An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization, Design Automation for Embedded Systems (2008)
11. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism, http://www.trimaran.org
12. Xilinx Data Sheet: Virtex 2.5V FPGA Detailed Functional Description, DS003-2, Version 2.8.1, (2002)
13. The Embedded Microprocessor Benchmark Consortium: http://eembc.org
14. Lee C., Potkonjak M. and Mangione-Smith W.H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 330-335 (1997)
15. Guthaus M.R., Ringenberg J.S., Ernst D., Austin T.M., Mudge T. and Brown R.B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite, IEEE International Workshop on Workload Characterization, pp. 3-14 (2001)