Runtime Reconfiguration of Custom Instructions for Real-Time Embedded Systems

Huynh Phung Huynh and Tulika Mitra
School of Computing
National University of Singapore
{huynhph1,tulika}@comp.nus.edu.sg

***Abstract*—This paper explores runtime reconfiguration of custom instructions in the context of multi-tasking real-time embedded systems. We propose a pseudo-polynomial time algorithm that minimizes processor utilization through customization and runtime reconfiguration, while satisfying all the timing constraints. Our experimental infrastructure consists of Stretch customizable processor supporting runtime reconfiguration as the hardware platform and realistic embedded benchmarks as applications. We observe that runtime reconfiguration of custom instructions can help to reduce the processor utilization by up to 64%. The experimental results also demonstrate that our algorithm is highly scalable and achieves optimal or near optimal (3% difference) processor utilization.**

I. Introduction

Application-specific customizable processor cores enable the design of complex, high-performance, low-power embedded systems under tight time-to-market constraints. Customizable processors are configurable w.r.t. the micro-architectural parameters. More importantly, they support extension of the core instruction set architecture with application-specific custom instructions. Custom instructions encapsulate the frequently occurring computation patterns in an application. They are implemented as custom functional units (CFU) in the datapath of the existing processor core. CFUs improve performance and reduce energy consumption of the applications. Lx, ARC^{TM} core, Xtensa, and Stretch S5 are some examples of commercial customizable processors.

As the total available area for implementation of the CFUs is limited, we may not be able to exploit the full potential of all the custom instructions in an application. This is particularly true if the application consists of a large number of kernels and each kernel requires unique custom instructions — a scenario that is quite common in high-performance embedded systems. In this context, runtime reconfiguration of the CFU fabric appears quite promising [15]. Here the set of custom instructions implemented in the fabric can change over the lifetime of the application. For multi-kernel applications, runtime reconfiguration is specially attractive, as the fabric can be tailored to implement *only* the custom instructions required by the active kernel(s) at any point of time. Of course, this virtualization of the CFU fabric comes at the cost of reconfiguration delay. The designer has to strike the right balance between the number of configurations and the reconfiguration cost.

In this paper, we explore runtime reconfiguration of custom instructions in the context of multi-tasking real-time embedded systems. More concretely, we assume that the application is specified as a set of task graphs (consisting of a number of tasks with dependencies among them), each associated with a period and a deadline. We only consider static non-preemptive schedules. Our objective is to minimize the processor utilization through appropriate selection of custom instructions for each task and the reconfiguration points while ensuring that all the timing constraints are satisfied. The custom instructions, in this scenario, may enable an application that was originally not schedulable to meet its deadlines by reducing execution times. Second, a lower processor utilization can exploit voltage scaling opportunities to lower the energy consumptions and still meet the computational requirements of the application.

II. Related Work

Many custom instructions generation techniques have been proposed in the literature, for example [1], [3], [16] among others. However, none of these approaches targets platforms exploiting dynamic reconfiguration of custom functional units. Recently, Bauer et al. [2] have developed rotating instruction set processing platform that selects custom instructions at runtime. We [9] have earlier presented an efficient framework for runtime reconfiguration of custom instructions. However, the above studies do not consider real-time constraints.

Co-synthesis of periodic task graphs with real-time constraint onto heterogeneous distributed embedded systems is addressed in [5], [11]. [7] partitions a task graph with timing constraints into a set of hardware units. Enforcing schedulability of real-time tasks with hardware implementation appears in [14], while [8] studies instruction-set customization for real-time tasks. None of these techniques takes into account the reconfiguration overhead or possibility of both spatial and temporal partitioning. [6], [12] co-synthesize real-time task graphs onto distributed systems containing dynamically reconfigurable FPGAs. These works assume a single hardware implementation of a task in FPGA and do not explore the hardware design space to evaluate tradeoffs between different implementations of the same task. Moreover, they do not put any hardware area constraint and try to minimize either cost (area), power or tardiness function while real-time constraints are satisfied. Finally, majority of the work in temporal partitioning comes from the reconfigurable computing community [4], [13], [10] . However, none of these studies considers real-time constraint. Our solution takes into consideration both dynamic (runtime) reconfiguration for instruction-set extensible processors and real-time constraints.

III. Problem Formulation

We model the application as a set of periodic task graphs (refer Figure 1), which has been widely used in previous works

978-3-9810801-5-5/DATE09 © 2009 EDAA

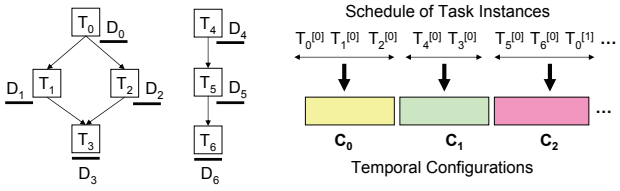

Fig. 1. A set of periodic task graphs and its schedule

[5], [6], [11], [12]. Each task graph is a directed acyclic graph consisting of a number of tasks. Let $\{T_0, \ldots, T_{N-1}\}$ be the set of N tasks corresponding to all the task graphs. A directed edge between two tasks $T_i \rightarrow T_j$ in a task graph denotes that task T_j can start execution only after task T_i completes execution. Let e_i denote the execution time of T_i in software, i.e., without any optimization through custom instructions. Each task graph has a deadline less than or equal to its period. The deadline D_i of T_i is the latest finish time of T_i derived from a backward topological search of the corresponding task graph starting with the sink node (whose deadline coincides with the task graph deadline).

The underlying processor platform allows optimized implementation of the tasks by exploiting custom instructions. Multiple custom instruction-set (CIS) versions are generated for each task with a trade-off between hardware area and performance gain. A CIS version consists of a set of custom instructions extracted from the corresponding task under an area constraint. In general, the performance gain of a CIS version increases with larger area. Let $\{v_i^0, \ldots, v_i^{M_i}\}$ denote the possible CIS versions of task T_i. In addition, let g_i^k and a_i^k denote the performance gain and area requirement of the version v_i^k. We assume v_i^0 corresponds to the software implementation, i.e., $g_i^0 = 0$ and $a_i^0 = 0$. In other words, for each task T_i, we have a choice of one software implementation and M_i implementations accelerated with custom instructions. The area A available for implementation of the CFUs can be reconfigured at runtime to support a different set of custom instructions. In this work, we focus on inter-task reconfiguration and do not consider intra-task reconfiguration. So the CIS version of a task must fit into the available area without reconfiguration, i.e., $a_i^k \leq A$.

Our objective is to come up with a static non-preemptive schedule of the task set that minimizes processor utilization by exploiting (a) processor customization and (b) runtime reconfiguration of the custom instructions, while satisfying deadline constraints. We need to construct our static schedule for the *hyper-period (HP)*, which is the least common multiple of the task graph periods. All the tasks in a task graph have the same period. Let P_i denote the period of task T_i. Clearly, a task T_i has $\frac{HP}{P_i}$ instances within the hyper-period. The s^{th} instance of T_i, denoted as $T_i^{[s]}$, has the deadline

$$deadline(T_i^{[s]}) = D_i + s \times P_i \tag{1}$$

In a feasible schedule, all the task instances meet their deadlines.

To minimize processor utilization, we need to assign appropriate CIS version to each task instance in the schedule. However, as we can exploit runtime reconfiguration of the custom instructions, we need not restrict ourselves to the area constraint A. Instead, we can perform *temporal partitioning* of the schedule into C configurations, where area constraint A is imposed on each configuration. For example, Figure 1 illustrates an initial portion of the schedule and its partitioning into three configurations. Note that each configuration contains a disjoint subsequence of task instances from the original schedule. Temporal partitioning allows us to work with a larger *virtual area* at the cost of a delay ρ per reconfiguration. The area A within a configuration is *spatially partitioned* among the task instances assigned to it by choosing appropriate CIS version for each task instance.

A feasible solution to this problem is a static, non-preemptive schedule of the task instances over the hyper-period where (a) the schedule is partitioned into C configurations, (b) each task instance is assigned to an appropriate CIS version, (c) the total area requirement of the chosen CIS versions within a configuration satisfies area constraint A, (d) each task instance satisfies its deadline constraint given by Equation 1, and (e) task dependence constraints are satisfied. The processor utilization U over the hyper-period HP for this solution can be expressed as

$$U = \frac{\left(\sum_{i=0}^{N-1} \frac{HP}{P_i} \times e_i\right) - \left(\left(\sum_{i=0}^{N-1} \sum_{s=0}^{\frac{HP}{P_i}-1} gain(T_i^{[s]})\right) - \rho * (C-1)\right)}{HP} \tag{2}$$

where $gain(T_i^{[s]})$ is the performance gain of the s^{th} instance of task T_i based on its assigned CIS version. As stated before, our objective is to construct the solution that minimizes U. In other words, we try to maximize the performance gain minus the reconfiguration cost

$$maximize \quad \left(\sum_{i=0}^{N-1} \sum_{s=0}^{\frac{HP}{P_i}-1} gain(T_i^{[s]})\right) - \rho * (C-1) \tag{3}$$

IV. Algorithm

The problem defined in Section III consists of two subproblems, namely, task scheduling and CIS version assignment. Design of optimal task scheduling algorithm is not the focus of this work. Instead, we employ *list scheduling* and use deadlines of task instances as the scheduling priority, i.e., a task instance with earlier deadline has higher priority. Still temporal partitioning of the resulting schedule into multiple configurations and assigning appropriate CIS versions to the task instances within each configuration with the objective of minimizing processor utilization (Equation 2), while satisfying all deadline constraints (Equation 1) is a non-trivial problem. In this section, we present an elegant solution based on dynamic programming.

List scheduling employed on the task graphs (as shown in Figure 1) over the hyper-period constructs a linear schedule of the task instances with possible idle periods in between. Let $\langle T_0, T_1, \ldots, T_X \rangle$ be the resulting schedule of task instance where $X = \sum_{i=0}^{N-1} \frac{HP}{P_i}$. *For simplicity of exposition, we ignore the superscripts for task instances in the rest of the paper.* If all the task instances can meet their deadlines in this schedule

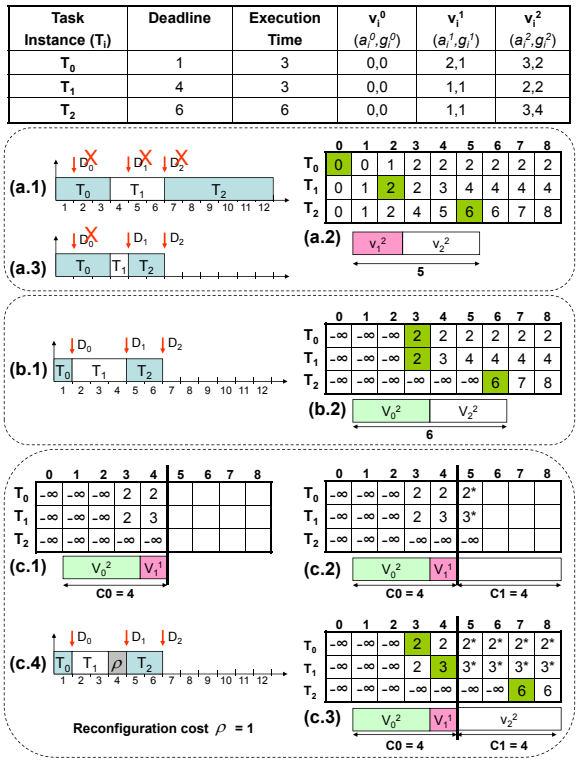

Fig. 2. Running Example

without any hardware acceleration, then we can guarantee that reduction in execution time of a task with custom instructions will still maintain schedulability. The problem gets a little simplified in this case. But if some of the task instances fail to meet deadlines, then our first priority is to ensure schedulability through hardware acceleration.

Running Example: Throughout this section, we use a simple example to illustrate our algorithm and convey the intuition behind it. Let us assume a schedule consisting of three task instances $\langle T_0, T_1, T_2 \rangle$. The deadlines D_i and execution times e_i of the software implementation of the tasks appear in the table in Figure 2. Clearly, all the tasks will miss their deadlines with the software implementations as shown in Figure 2 (a.1). Therefore, we would like to explore processor customization so as to reduce execution times of the tasks and meet the deadlines. The table in Figure 2 also shows that each task T_i has three CIS versions v_i^0, v_i^1, v_i^2 with varying area and performance gain (e.g., 3,2 for v_0^2 denotes 3 units of area and 2 units of performance gain). The version v_i^0 is the software implementation (zero hardware area and zero performance gain).

A. A Simple Solution

Let us for the moment ignore reconfiguration cost, configuration boundaries, and deadline constraints. Our objective is to find an assignment of CIS versions to the tasks to achieve maximum performance gain (given by Equation 3) under a *virtual area* constraint. Given a virtual area *area*, let us define the maximum performance gain of the sequence $\langle T_0, \ldots, T_i \rangle$ as $G_i(area)$. If we ignore reconfiguration cost and configuration boundaries, we can compute $G_i(area)$ for different values of *area* through dynamic programming. $G_i(area)$ can be defined recursively as

$$G_i(area) = \max_{\substack{k=0,\ldots,M_i \\ a_i^k \leq area}} \left(g_i^k + G_{i-1}(area - a_i^k) \right) \tag{4}$$

That is, given a virtual area *area*, we explore all possible CIS versions of T_i and choose the one that results in maximum performance gain for $\langle T_0, \ldots, T_i \rangle$. The base case for task T_0

$$G_0(area) = \max_{\substack{k=0,\ldots,M_0 \\ a_0^k \leq area}} g_0^k \tag{5}$$

Example: Figure 2 (a.2) shows the performance gains for the tasks under area constraints 0 to 8 where 8 is the area required to implement the best CIS versions of all the tasks. Total execution time of T_0, T_1, T_2 in software is 12 time units whereas the entire sequence should complete execution within 6 time units. The solution table indicates that we can obtain a performance gain of 6 time units for the task sequence with 5 units of area. The solution cells corresponding to this performance gain are shaded in Figure 2 (a.2); the CIS versions chosen are v_0^0, v_1^2, and v_2^2. Unfortunately, the first task T_0 fails to meet its deadline because it is implemented purely in software as shown in Figure 2 (a.3) (execution time = 3 while deadline = 1). This example clearly shows that we cannot ignore the deadline constraints of the individual tasks (T_0 and T_1) while constructing the solution to maximize performance gain.

B. Deadline Constraints

The recurrence defined by Equation 4 does not take into account the deadline constraints. Let us now proceed to modify this equation so as to maximize performance gain while satisfying deadline constraints. We will continue to ignore reconfiguration at this point.

Given a virtual area constraint *area*, we find the solution with the maximum performance gain $G_i(area)$ and each task $T_0, \ldots, T_i$ is assigned one of its CIS versions. The solution is *feasible* if all the tasks $T_0, \ldots, T_i$ can meet their deadlines with the CIS version assignments in the solution. To satisfy the deadline constraints, we modify the construction of dynamic programming solution table with the following consideration. While exploring CIS versions of task T_i, we need to choose the solution that returns best $G_i(area)$ and $T_0, \ldots, T_i$ meet their deadlines. So how do we impose this constraint? Equation 4 is now modified as

$$G_i(area) = \max_{\substack{k=0,\ldots,M_i \\ a_i^k \leq area \\ is\_schedulable(T_i)}} \left(g_i^k + G_{i-1}(area - a_i^k) \right) \tag{6}$$

Here, is_schedulable(T_i) simply checks the deadline constraints for $T_0, \ldots, T_i$. In fact, as we ensure the sequence $\langle T_0, \ldots, T_{i-1} \rangle$ is already schedulable, we only need to check that T_i meets its deadline. If we cannot find any CIS version assignment for T_i to make the sequence $\langle T_0, \ldots, T_i \rangle$ schedulable, we set $G_i(area) = -\infty$.

Example: In our example, the feasible schedule is shown in Figure 2 (b.1) and the solution table is shown in Figure 2 (b.2). When $area < 3$, $G_0(area) = -\infty$ which shows T_0 misses its deadline. Clearly, $G_1(area)$, $G_2(area)$ are also equal to $-\infty$ when $area < 3$. The difference between Equation 4 and Equation 6 becomes clear by looking at the last row in Figure 2 (b.2). For example, when $area = 5$, we cannot find any CIS version for T_2 to make it schedulable and $G_2(5) = -\infty$ instead of $G_2(5) = 6$ in Figure 2 (a.2). The shaded cells in Figure 2 (b.2) provide the optimal solution that satisfies all the deadline constraints. Here T_0 selects v_0^2, T_1 selects v_1^0 (implemented in software) and T_2 selects v_2^2.

C. Runtime Reconfiguration

So far we assume that the entire virtual area is available as a single continuous configuration. However, in reality, the virtual area is divided into a number of configurations and reconfiguration cost is incurred while switching from one configuration to another. Suppose the area constraint for a single configuration $A = 4$ in our example. Let us now investigate the optimal solution returned in the previous subsection where T_0 selects v_0^2 (3 unit area), T_1 selects v_1^0 (implemented in software) and T_2 selects v_2^2 (3 unit area) in Figure 2 (b.2). This solution is no longer feasible for the following reasons

- A task instance should be mapped to only one configuration; it cannot straddle across configuration boundaries. In our example, task T_2 occupies 1 unit of area in the first configuration and 2 units of area in the second configuration.
- The reconfiguration cost should be taken into account while computing performance gain.

Restricting a task instance to one configuration: How do we handle the constraint that a task cannot straddle across configuration boundaries? Given a virtual area *area*, the number of configurations is $C = \lceil \frac{area}{A} \rceil$ and the area available in the last configuration *physical_area* is

$$physical\_area = \begin{cases} A & \text{if } area \bmod A = 0 \\ area \bmod A & \text{otherwise} \end{cases} \quad (7)$$

When exploring the CIS versions of task T_i under area constraint *area*, we should now impose the constraint that the available area is less than the *physical_area*, i.e., the area of the current configuration. We modify Equation 6 to reflect this.

$$G_i(area) = \max_{\substack{k=0,\ldots,M_i \\ a_i^k \le physical\_area \\ is\_schedulable(T_i)}} \left(g_i^k + G_{i-1}(area - a_i^k) \right) \quad (8)$$

Reconfiguration cost: We now need to subtract the reconfiguration cost from the total performance gain under the following conditions.

1) If the area requirement of a CIS version is equal to the area of the current configuration, i.e., $a_i^k = physical\_area$ and $C > 1$, then T_i is the *first* task in the current configuration. We should subtract the reconfiguration cost from the gain.

2) The reconfiguration cost offsets the performance gain of the CIS version chosen for task T_i. Hence, $T_0, \ldots, T_i$ may have obtained greater performance gain when reconfiguration was not involved. That is, it is possible to have $G_i(area) \le G_i(area - physical\_area)$. In this case, it does not make sense to perform reconfiguration before task T_i and we should instead select the solution with gain $G_i(area - physical\_area)$. Even if $G_i(area)$ is equal to $G_i(area - physical\_area)$, we still prefer the solution $G_i(area - physical\_area)$ as it is better not to use the current configuration, if possible. The fact that a solution does not use the current configuration is represented visually with a '*' in Figure 2 and maintained as a binary variable $reconfig_i(area)$. If the solution for tasks $T_0, \ldots, T_i$ under *area* has not used any portion of the current configuration, then $reconfig_i(area) = false$; otherwise we set $reconfig_i(area) = true$.

3) Suppose in Equation 8, we use the partial solution $G_{i-1}(area)$ where $reconfig_{i-1}(area) = false$ (marked with a *), i.e., the solution did not use the current configuration. If we combine this solution with a CIS version of T_i, the implication is that T_i is the first task to use the current configuration. Therefore, reconfiguration cost should be subtracted from the total performance gain. The modification of Equation 8 to take reconfiguration cost into account is easier to present in an algorithmic form as shown in Algorithm 1.

Algorithm 1: Compute $G_i(area)$

```
1  C ← ⌈area / A⌉;
2  physical_area ← { A              if area mod A = 0
                   { area mod A     otherwise
3  G_i(area) ← −∞;  reconfig_i(area) ← false;
4  for k = 0 to M_i do
5      if a_i^k ≤ physical_area then
6          gain ← g_i^k + G_{i−1}(area − a_i^k);
7          reconifguration ← false;
8          if C > 1 AND (a_i^k = physical_area OR !reconfig_{i−1}(area − a_i^k)) then
9              gain ← gain − ρ;  reconfiguration ← true;
10         if is_schedulable(T_i) AND gain > G_i(area) then
11             G_i(area) ← gain;  reconfig_i(area) ← reconfiguration;
12 if C > 1 AND G_i(area) ≤ G_i(area − physical_area) then
13     G_i(area) ← G_i(area − physical_area);  reconfig_i(area) ← false;
```

Example: Now let us get back to our running example. Tasks T_0, T_1, T_2 cannot have a feasible solution when we restrict ourselves to one configuration (4 units of area) and take schedulability constraints into account (Figure 2 (c.1)). Let us now look at performance gain with 5 units of area in Figure 2 (c.2). Task T_0 cannot obtain any further performance gain. Therefore, its solutions is marked with '*' in the second configuration indicating that T_0 belongs to the previous configuration.

The situation gets interesting with T_1. If T_1 is implemented in the second configuration, it can get a maximum gain of 2 time units. However, we need to subtract reconfiguration cost of 1 time unit. As T_0 has a gain of 2 time units, the total performance gain for T_0, T_1 with two configurations is only $2 + 2 - 1 = 3$. On the other hand, we can easily get a gain of 3 units by implementing both T_0 and T_1 in the first configuration as shown by the shaded cells in Figure 2 (c.3). Therefore, it does not make sense to put T_1 into the second configuration and its cell is marked with '*'.

Finally, T_2 fails to meet its deadline in the beginning by using the second configuration as reconfiguration cost overshadows the performance gain. However, when $area = 7$, T_2 can implement its best CIS version in the second configuration with 4 units of performance gain (Figure 2 (c.3)). T_0, T_1 gets 3 units of gain from the first configuration. Therefore, total performance gain is $3+4-1=6$. At this point, we have been able to construct a solution that satisfies all the timing constraints as shown in Figure 2 (c.4).

D. Putting It All Together

We can now present our complete dynamic programming (called *DP*) algorithm (Algorithm 2) that satisfies deadline constraints as well as takes into account runtime reconfiguration.

Algorithm 2: Maximize Performance Gain

```
1 for area = Δ to Max_A in steps of Δ do
2     for i=0 to X − 1 do
3         if ∀j ≤ i  !reconfig_j(⌊area/A⌋ × A) then
4             G_i(area) = G_i(⌊area/A⌋ × A);
5         else
6             compute G_i(area);
7     if area mod A = 0  AND  ∀ i  !reconfig_i(area) then
8         break;
9 return G_{X−1}(area);
```

Let $X = \sum_{i=0}^{N-1} \frac{HP}{P_i}$ be total number of task instances over the hyper-period. We vary *area* in steps of Δ to the area required to implement the best CIS versions of all task instances *Max_A*. For each *area*, we do not compute $G_i(area)$ if performance gains of $\langle T_0, \ldots, T_i \rangle$ have no improvement compared to the preceding configuration $\left(\lfloor \frac{\text{area}}{\text{A}} \rfloor \times \text{A}\right)$ (line 3). Therefore, $G_i(area)$ should be filled up with the solution from the preceding configuration (line 4) as performance gain is guaranteed to have no improvement in the current configuration either. In Figure 2 (c.3), performance gains of $\langle T_0, T_1 \rangle$ have no improvement in configuration C_1. Therefore, the performance gain of $\langle T_0, T_1 \rangle$ will remain unchanged in all the future configurations. We compute $G_i(area)$ through Algorithm 1 (line 6). Finally, if performance gains of $\langle T_0, \ldots, T_{X-1} \rangle$ have no improvement at the end of the current configuration, we will stop the algorithm (lines 7-8). This is because we cannot get any additional performance gain by exploring further configurations.

Algorithm Complexity: For each task instance, we compute $G_i(area)$ (Algorithm 1) with $area = 0 \ldots Max\_A$ in steps of Δ. Moreover, for each $G_i(area)$ we have $(M_i + 1)$ choices of CIS version. Let $M_{max} = \max_{i=0 \ldots N-1}(M_i + 1)$. Therefore, the worst case complexity of our algorithm is $O(X \times \frac{Max\_A}{\Delta} \times M_{max})$.

V. Experimental Evaluation

Each task graph (see Figure 3) used in our experiment combines real kernels from the same application domain to form meaningful benchmarks, such as JPEG decoder (TG1) and encoder (TG4), automotive application (TG3), and consumer electronic applications (TG0, TG2, TG5).

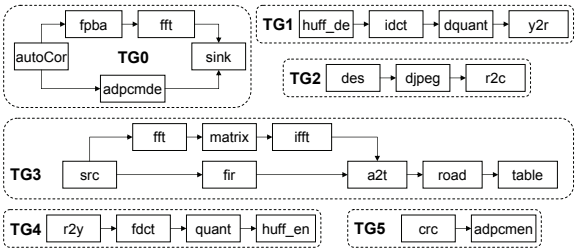

Fig. 3. Task Graphs

Given each task, custom instruction versions are manually generated for the Stretch S5 platform [15] by using Stetch C language. We can achieve different custom instruction versions (or CIS versions) by changing the unroll factor of the compute-intensive loops within the task or the number of custom instructions. The higher unroll factor results in larger hardware area requirement and better performance gain. The profiler in Stretch can provide us the performance gain and hardware area of the CIS versions of each task.

We create four combination of task graphs, *A0*, *A1*, *A2*, *A3*, each consisting of two to four task graphs from Figure 3 to represent different applications. *A0*, *A1*, *A2*, *A3* consist of {TG1,TG4,TG5}, {TG1,TG3}, {TG0,TG2,TG4,TG5}, and {TG2,TG4,TG5} respectively. To set the periods for the task graphs, we choose a total processor utilization U for the entire system (without any custom instructions) and then select the period for each constituent task graph to achieve the corresponding utilization. We vary U between $0.9 - 1.4$ for each scenario. $U > 1$ implies that the application scenario is definitely not schedulable without custom instructions, whereas it may or may not be schedulable with $U \leq 1$.

The configuration time of the whole CFU fabric of Stretch, which includes 4096 4-bit AUs and 8192 4-bit $\times$ 8-bit MUs, is approximately 100μs or roughly 30K CPU cycles at 300MHz core. We define one hardware area unit to be a tuple of 400 AUs and 800 MUs. As configuration time is proportional to fabric size, configuration time of one hardware area unit is approximately 3K CPU cycles. For each application, we vary the CFU fabric size between 10-100% (in steps of 10%) of the maximum area required to implement the best CIS versions of the constituent kernels, *Max_A*. When maximum area is available, an application explores the limit of speedup achievable though custom instructions without reconfigurations.

Given an application scenario, area constraint and processor utilization, we apply three different techniques to generate a feasible schedule and CIS assignments with minimum processor utilization: (1) our **DP** algorithm proposed in Section IV. (2) **Optimal:** an Integer Linear Programming (ILP) formulation that can return the optimal solution. We note that the optimal ILP formulation is non-trivial as it includes task scheduling, CIS version assignment, and runtime reconfiguration. However, due to space constraints, we do not include the details here. (3) **Static:** this solution restricts itself to a static configuration, i.e., it does not consider dynamic reconfiguration. This is a simplified version of the ILP formulation for *Optimal* that excludes dynamic reconfiguration.

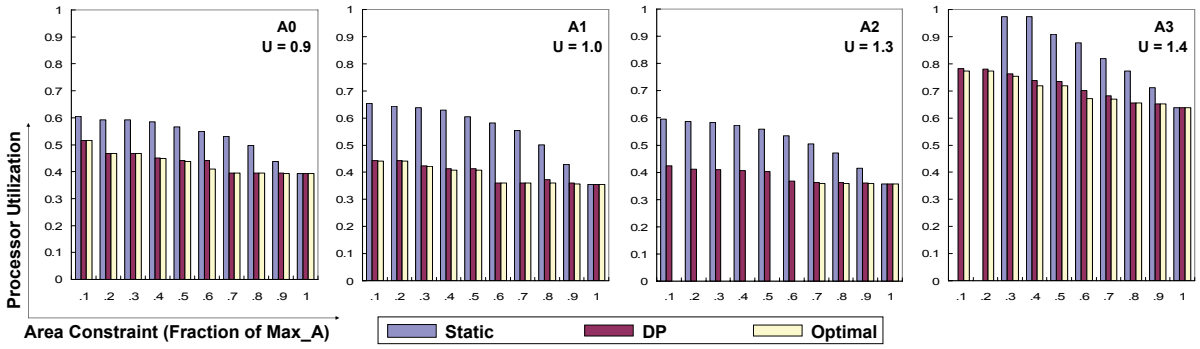

Fig. 4. Comparison of DP, Optimal, and Static

Figure 4 shows the accuracy of our algorithm *DP* compared to *Optimal*. This figure also shows the advantage of runtime reconfiguration (*Optimal* and *DP*) over static configuration (*Static*). Due to space constraints, we only show the results for one particular input processor utilization U for each application scenario. *DP* achieves up to 37% better processor utilization compared to *Static* when area constraint decreases. This is expected as runtime reconfiguration can fit more custom instructions into the fabric through temporal sharing. Note that for the application *A3*, when the area constraint is really tight, i.e. $0.1 * Max\_A$, there does not exist any feasible solution with static configuration *Static*. But feasible solutions can be obtained with runtime reconfiguration. More importantly, the solution returned by *DP* often coincides with the optimal solution. In fact it is mostly within 3% of the optimal processor utilization. Moreover, for the application *A2*, when area constraint is small, we do not get *Optimal* result as ILP solver fails to return any solution.

Schedule Length	Task Graph Sets	Optimal	DP
11	*A3*	19.525920	0.179993
12	*A0*	94.245572	0.246194
13	{TG0,TG2,TG5}	168.509196	0.492661
14	{TG2,TG5}	193.335448	1.182449
15	{TG0,TG1,TG4,TG5}	1273.653911	0.795110
16	*A2*	N/A	0.350204
17	{TG0,TG5}	N/A	0.903613
18	{TG0,TG1,TG2,TG4,TG5}	N/A	1.513959

TABLE I
RUNNING TIME OF OPTIMAL AND DP IN SECONDS.

Running times of both *Optimal* and *DP* depend on the number of task instances in the schedule, *schedule length*. {*A0,A1,A3*} have schedule lengths 11 or 12 while *A2* has schedule length 16. To show the effect of schedule length on running time of both algorithms, we create more task graph sets with schedule lengths varying from 11 to 18. Table I shows running times of *Optimal* and *DP* on different schedule lengths when input processor utilization is $U = 1$ and area constraint is $0.3 * Max\_A$ for different task graphs. The running time of *Optimal*, while relatively small with schedule lengths of {11,12}, shoots up quickly at schedule length 16. The solution for *Optimal* cannot be obtained even after waiting for two days when schedule lengths are greater than 16. Clearly, *DP* is significantly more scalable compared to *Optimal*.

VI. CONCLUSIONS

We propose a pseudo-polynomial time algorithm to efficiently solve the problem of runtime reconfiguration of custom instructions for real-time embedded systems. Minimized processor utilization is achieved through appropriate custom instructions selection as well as temporal partitioning with consideration of reconfiguration cost. Our experiments using real embedded benchmarks on Stretch customizable processor show scalability and accuracy of our algorithm compared to integer linear programming based optimal solutions.

VII. ACKNOWLEDGEMENTS

This work is partially supported by NUS research project R-252-000-292-112.

REFERENCES

[1] K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk. CHIPS: Custom hardware instruction processor synthesis. In *IEEE TCAD'08*.
[2] L. Bauer, M. Shafique, S. Kramer, and J. Henkel. RISPP: Rotating instruction set processing platform. In *DAC '07*.
[3] P. Bonzini and L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *DATE '07*.
[4] K. S. Chatha and R. Vemuri. Hardware-software codesign for dynamically reconfigurable architectures. In *FPL '99*.
[5] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-software co-synthesis of embedded systems. In *DAC '97*.
[6] R. P. Dick and N. K. Jha. CORDS: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *ICCAD '98*.
[7] T. Hollstein, J. Becker, A. Kirschbaum, and M. Glesner. HiPART: A new hierarchical semi-interactive HW-/SW partitioning approach with fast debugging for real-time embedded systems. In *CODES/CASHE '98*.
[8] H. P. Huynh and T. Mitra. Instruction-set customization for real-time embedded systems. In *DATE '07*.
[9] H. P. Huynh, J. E. Sim, and T. Mitra. An efficient framework for dynamic reconfiguration of instruction-set customization. In *CASES '07*.
[10] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In *DAC '99*.
[11] Y. J. Kim and T. Kim. HW/SW partitioning techniques for multi-mode multi-task embedded applications. In *GLSVLSI '06*.
[12] B. Mei, P. Schaumont, and S. Vernalde. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *ProRISC '00*.
[13] K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers '99*.
[14] Y. Shin and K. Choi. Enforcing schedulability of multi-task systems by hardware-software codesign. In *CODES '97*.
[15] Stretch. S5000 software-configurable processors, 2004.
[16] P. Yu and T. Mitra. Scalable custom instruction identification for instruction-set extensible processors. In *CASES*, 2004.