Customized MPSoC Synthesis for Task Sequence

Liang Chen
Department of Computer Science
National University of Singapore
chenliang@nus.edu.sg

Nicolas Boichat
Department of Computer Science
National University of Singapore
nboichat@nus.edu.sg

Tulika Mitra
Department of Computer Science
National University of Singapore
tulika@comp.nus.edu.sg

***Abstract*—Multiprocessor System-on-Chip (MPSoC) platforms have become increasingly popular for high-performance embedded applications. Each processing element (PE) on such platforms can be tuned to match the computational demands of the tasks executing on it, creating a heterogeneous multiprocessor system. Extensible processor cores, where the base instruction-set architecture can be augmented with application-specific custom instructions, have recently emerged as flexible building blocks for heterogeneous MPSoC platforms. However, the customization of the different PEs has to be carried out in a synergistic manner so as to create an optimal system. In this work, we propose a pseudo-polynomial time algorithm to design the most resource-efficient customized MPSoC platform for mapping linear task graphs representing streaming applications, under deadline constraints. Experimental validation with MP3 encoder and MPEG-2 encoder applications confirms the efficiency of our approach.**

I. Introduction

Multiprocessor System-on-Chip (MPSoC) platforms are being increasingly deployed in high-performance embedded systems. MPSoC platforms employ heterogeneous processing elements (PEs) to construct a system that perfectly matches the application requirements, leading to significant cost and power savings. The processing elements can be diverse in nature and may include general purpose processors, DSPs, application-specific hardware accelerators, and others. However, system integration and lack of flexibility are some of the major challenges faced by the designers in creating platforms with such disparate architectures.

Extensible processor cores (e.g., Xtensa from Tensilica [1]), where the existing base instruction-set architecture (ISA) can be enhanced with application-specific custom instructions, are emerging as promising alternatives in this context. Custom instructions encapsulate frequently executed computational patterns in the application. They are implemented as hard-wired datapaths (custom functional units) in the existing processor core and help improve the power consumption and performance of the application. A heterogeneous MPSoC may consist of a number of extensible processor cores, where each core has been customized according to the application requirements. As all the PEs share the same base ISA and a common core, application development on such platforms is relatively straightforward.

MPSoC platforms consisting of extensible processor cores are an excellent match for streaming applications [2], [3]. These applications can be partitioned into multiple compute-intensive kernels or tasks and represented in the form of an acyclic task graph. For example, Figure 1 show the task graphs corresponding to MP3 encoder and MPEG-2 encoder applications. As the applications are expected to process continuous stream of data, the tasks can execute in a pipelined fashion, where different tasks can process data from different iterations in parallel.

The goal of our work is to synthesize an optimal customized MPSoC platform for a given streaming application. Notice that this optimization involves two conflicting objectives of minimizing the resource requirement and minimizing the pipeline period (or equivalently maximizing the throughput). Hence given a period constraint imposed by the system designer (e.g., for MPEG-2 encoder, 30 frames per second would be the minimum speed to provide smooth viewing experience), we are interested in minimizing the resource requirement while satisfying this period constraint. The area requirement includes the base area of the processing cores as well as the area for the custom functional units. Mapping the tasks to PEs and the customization of each PE can dramatically influence the area and period of the entire system. Therefore, our design space exploration algorithms, tune the processors in a synergistic manner to create optimal systems.

Compared to state-of-the-art approaches, we show an efficient hierarchical algorithm that separates task mapping and custom instruction sets selections, and returns optimal solutions. Rather than focusing on fixed architectures with a given number of PEs [4], [5], or performing an one-to-one mapping of tasks to PEs [2], [3], we consider different number of PEs and interval-based mapping policy. Observing the design space for MP3 or MPEG-2 encoder shown in Figure 3, it should be obvious that fixing the number of PEs can easily miss the global optima. Most importantly, rather than using a heuristic, we design a pseudo-polynomial time algorithm that returns the optimal solution in a fraction of the time required by an exhaustive approach.

II. Related Work

We use a number of Tensilica LX2 [6] processor cores, enhanced with custom instructions as the MPSoC platform, which is similar to architectures used in [3], [5].

In terms of task mapping, Benoit et al. [7] classify the policies to map tasks onto a fixed number of PEs into three categories: one-to-one mapping, where each task gets its own dedicated PE [3], [2], an interval-based policy, where only tasks that are contiguous in the task graph can be mapped on

a single PE [5], and a fully general policy without restrictions [4]. In this work, we use the interval-based policy, on a variable number of PEs.

Our interval based approach for load balancing among PEs could be modeled as a chain-on-chain problem (CCP) [8]. The CCP problem has been widely studied, and various efficient polynomial time algorithms have been proposed [9], [10], [11]. However, these algorithms mainly focus on homogeneous processors, with the goal of maximizing the throughput. Our approach focuses on heterogeneous multi-processors, enhanced with task-specific custom instruction sets. Besides, the goal is modified to minimize the area consumption under a designer-imposed throughput constraint.

The simpler problem of mapping tasks onto processors using different operating frequencies is NP-complete for interval-based policy, fully general policy and one-to-one mapping with heterogeneous communication costs [7]. Thus several works propose algorithms that approximate an optimal solution: [5] proposes an iterative heuristic approach, [4] uses evolutionary algorithms, and [3] uses heuristic approach as well. On the other hand, [2] proposes an integer linear programming (ILP) formulation, assisted with safe heuristics, which guarantees an optimal solution. Although [2] also focuses on optimal solutions, they only consider one-to-one mappings and a fixed number of PEs. Our approach finds the optimal solution considering interval-based task mapping and various PE numbers. Adapting our approach to a fully general mapping policy, which should take tasks clustering into account [12], [13], is left as future work.

III. Problem Definition

The input to our framework is a linear task graph modeling the application. Let $\langle T_1, T_2, \ldots, T_N \rangle$ be the N tasks in a linear task graph representing a streaming application. There are dependencies between consecutive tasks in this linear chain. Task T_{i+1} can start execution only after task T_i has completed execution for $1 \leq i < N$. Note that our framework is not limited to applications that can be modeled as linear task graphs. An application that is modeled with a general task graph can be easily transformed into a linear chain while respecting all the dependencies in the original task graph. The maximum tolerable period $period$ (or minimum throughput) requirement of the application is also provided as an input.

We assume that each task in the task graph can be accelerated with the help of custom instructions. There are multiple implementations or versions of each task corresponding to different choices of custom instructions. We call each such implementation a custom instruction set or CIS, which consists of a set of custom instructions. Each CIS is associated with an area requirement and an execution time. The area requirement captures the additional area required to implement the specific function units for the custom instructions. Increasing the area available allows more flexibility for the implementation and thereby reduces the execution time. Let $\{C_{i,0}, C_{i,1}, \ldots, C_{i,m_i}\}$ denote the different custom instruction sets corresponding to task T_i where $m_i + 1$ is the number of CISs for T_i. Let us also assume that $a_{i,j}$ is the additional area required and $t_{i,j}$ is the execution time for the CIS $C_{i,j}$. Moreover, we assume that $C_{i,0}$ is the software implementation version with $a_{i,0} = 0$ and $t_{i,0}$ is the software execution time. We order the rest of the CISs according to their area requirement. That is, $a_{i,0} < a_{i,1} < \cdots < a_{i,m_i}$ and as we only consider Pareto-optimal CISs, $t_{i,0} > t_{i,1} > \cdots > t_{i,m_i}$.

The application is mapped onto an underlying architecture consisting of a linear chain of P processing elements $(PE_1, \ldots, PE_P)$ where $P \leq N$. The PEs form the different pipeline stages of the application. We impose the constraint that only a consecutive sequence of tasks from the linear task graph can be mapped to a PE. This is known as interval-based mapping. In other words, the linear task graph is divided into P partitions $(S_1, \ldots, S_P)$ where each partition is a consecutive sequence of tasks in the task graph and partition S_i maps to PE_i for $1 \leq i \leq P$. The pipeline stage with the maximum execution time determines the period and the throughput.

We start with homogeneous multi-core architecture, that is, the base instruction-set architecture of all the P processing elements are identical. The base area of each PE is $areaPE$. However, each PE can be customized by adding CISs according to the tasks mapped to it. So the final solution is a heterogenous multiprocessor system-on-chip (MPSoC) customized and optimized for the target application. The goal of our optimization strategy is to minimize the total area requirement of the MPSoC solution while satisfying the period or throughput constraint of the application. Both the base area of the PEs as well as the selected CIS versions of the tasks determine the area requirement of an MPSoC solution. In other words, our design space exploration need to explore (a) the number of PEs P, (b) the partitioning of the task graph into P partitions, and (c) the CIS choice for each of the N tasks.

So our problem definition can be formally stated as follows: Given a linear task graph consisting of N tasks with multiple CIS versions for each task and period constraint *period*, find the number of PEs P, the CIS version for each of the N tasks, and P partitions of the linear task graph so that the maximum execution time of each PE is less than $period$ and the total area (the base area for P PEs and the additional area for all the selected CIS versions) for the MPSoC solution is minimized.

IV. Exhaustive Design Space Exploration

We first start with a simple algorithm that exhaustively enumerates the entire design space. This helps us to visualize the complex tradeoff between area and performance. We will follow it up with more efficient approaches that can identify the resource-optimal solution under period constraint.

The exhaustive algorithm recursively enumerates all possible choices for each task. It processes the tasks in their linear order starting with task T_1. For task T_i, we enumerate all possible choices for CIS. For each such choice of CIS, we consider two alternative mapping choices for T_i. The first choice is to map T_i to the current PE. The other alternative is to map T_i to a new PE, in which case we add the base area of a PE $areaPE$ to our cumulative area variable $area$.

Algorithm 1: Exhaustive Algorithm

```
P = 1;
area = areaPE;
time = 0;
period = 0;
Traverse(1,P,area,time, period);

procedure Traverse(i, P, area, time, period)
    for j = 1 to m_i do
        /* map task T_i to old PE */
        tempArea = area + a_{i,j};
        tempTime = time + t_{i,j};
        if tempTime > period then
            tempPeriod = tempTime;
        if i < N then
            Traverse(i + 1, P, tempArea, tempTime,
            tempPeriod);
        else
            plot {tempPeriod, tempArea};

        /* map task T_i to new PE */
        if i ≠ 1 then
            tempArea = area + a_{i,j} + areaPE;
            tempTime = t_{i,j};
            if tempTime > period then
                tempPeriod = tempTime;
            if i < N then
                Traverse(i + 1, P + 1, tempArea, tempTime,
                tempPeriod);
            else
                plot {tempPeriod, tempArea};
```

At each point, we keep track of the period of the application, that is, the processing element with the maximum execution time. Once we have reached the last task, we simply plot the area requirement and the period of the solution.

Note that it is trivial to modify Algorithm 1 to compute the area-optimal solution under the a particular $period$ constraint. In this case, we have to make sure that the execution time of any PE is always under the $period$ constraint. If the constraint is violated at some point, we can simply prune away the rest of the recursions for that partial solution. We also need to keep track of the global optimal solution obtained so far. Once we have reached the last task, we check if the area requirement of the solution is better than the optimal solution and update the optimal area accordingly.

The complexity of the exhaustive design space algorithm is $O(m^N \times 2^{N-1})$ where m is the average number of CIS versions per task.

V. Integer Linear Programming (ILP) Formulation

We now present an Integer Linear Programming (ILP) formulation of the problem so that we can obtain an optimal solution with the help of an off-the-shelf ILP solver. However, as we will observe in the experimental evaluation section, ILP formulation does not scale well with the number of tasks N. So we will present an alternative scalable approach next.

Let $x_{i,j}$ be a binary variable that denotes whether CIS version $C_{i,j}$ is selected for task T_i.

$$x_{i,j} = \begin{cases} 1, \text{ if } C_{i,j} \text{ is selected} \\ 0, \text{ otherwise} \end{cases}$$

For each task T_i, only one CIS version can be selected.

$$\sum_{j=0}^{m_i} x_{i,j} = 1$$

Let $y_{i,k}$ be a binary variable that denotes whether task T_i is mapped to PE_k.

$$y_{i,k} = \begin{cases} 1, \text{ if } T_i \text{ is mapped to } PE_k \\ 0, \text{ otherwise} \end{cases}$$

Each task is mapped to exactly one PE.

$$\sum_{k=1}^{N} y_{i,k} = 1$$

In the summation term we have implicitly defined the number of processing elements to be N. This is necessary to keep the formulation linear. The solution may contain processing elements which have no tasks mapped to them and have to be eliminated. The number of valid processing elements P can be defined as

$$\sum_{i=1}^{N} y_{i,k} - U \times z_k \leq 0; \ \sum_{i=1}^{N} y_{i,k} + 1 - z_k > 0$$

$$P = \sum_{k=1}^{N} z_k$$

where U is a large constant greater than N. z_k is a binary variable which is equal to 1 if there is any task mapped to PE_k and 0 otherwise.

There is one important constraint that is imposed by interval-based mapping approach adopted in our framework. Two consecutive tasks T_i and T_{i+1} should either be mapped to the same PE or mapped to two adjacent PEs. In other words, if task T_i is mapped to PE_k, then task T_{i+1} can only be mapped to either PE_k or PE_{k+1}.

$$\sum_{k=1}^{N} k \cdot y_{i+1,k} \geq \sum_{k=1}^{N} k \cdot y_{i,k}$$

$$\sum_{k=1}^{N} k \cdot y_{i+1,k} \leq 1 + \sum_{k=1}^{N} k \cdot y_{i,k}$$

The period constraint can be imposed as follows.

$$\sum_{i=1}^{N} \sum_{j=0}^{m_i} t_{i,j} \cdot x_{i,j} \cdot y_{i,k} \leq period$$

This is a non-linear constraint. To linearize this constraint, we define a new binary variable $v_{i,j,k}$ where

$$v_{i,j,k} = 1 \Leftrightarrow (x_{i,j} = 1) \ \ AND \ \ (y_{i,k} = 1)$$

This condition can be expressed in linear form as follows.

$$v_{i,j,k} \leq x_{i,j}; \quad v_{i,j,k} \leq y_{i,k}; \quad v_{i,j,k} \geq x_{i,j} + y_{i,k} - 1$$

Now the period constraint can be re-written as

$$\sum_{i=1}^{N}\sum_{j=0}^{m_i} t_{i,j} \cdot v_{i,j,k} \leq period$$

Our objective function is to minimize the total area required

$$\text{Total area} = \sum_{i=1}^{N}\sum_{j=0}^{m_i} a_{i,j} \cdot x_{i,j} + P \cdot \text{area}PE$$

The most area-efficient solution can be obtained by minimizing the objective function under the constraints.

VI. Dynamic Programming Algorithm

We now proceed to present a dynamic-programming based efficient algorithm that can compute, in pseudo-polynomial time, the area-optimal solution under a period constraint. The algorithm proceeds in two stages. In the first stage, we compute the minimal area required to map a subsequence of tasks on a PE such that the period constraint is not violated. In the second stage, we choose the best partitioning of the tasks.

A. Customization

The goal of this stage is to compute the area-optimal solution for a sequence of tasks mapping to a single PE under the period constraint. In other words, the total execution time of the tasks should be less than $period$ while the area requirement of their selected CIS versions should be minimal.

Algorithm 2: Compute $area_{s,e}$ for all s, e

```
for s ← 0 to N do
    for e ← s + 1 to N do
        found = FALSE;
        for A ← 0 to AREA do
            for j ← 0 to m_e do
                if (a_{e,j} ≤ A) then
                    time_{s,e}(A) = min(time_{s,e}(A),
                    time_{s,e-1}(A − a_{e,j}) + t_{e,j})
            if (time_{s,e}(A) ≤ period AND !found) then
                area_{s,e} = A;
                found = TRUE;
```

Algorithm 2 computes the area-optimal solution for each possible subsequence $T_{s+1}, \ldots, T_e$ mapped to a PE under the $period$ constraint. The execution time of the subsequence mapped to a PE can be defined as

$$time_{s,e} = \sum_{i=s+1}^{e}\sum_{j=0}^{m_i} t_{i,j} \cdot x_{i,j}$$

Note that according to our definition, $time_{s,e}$ corresponds to the execution time of the task subsequence $[T_{s+1}, T_{s+2}, \ldots, T_e]$. We assume that $time_{s,s} = 0$, which means that there is no task mapped to the PE. Similarly, we have $area_{s,s} = 0$. We can compute the minimum value of $time_{s,e}$ for all possible values of s, e under different area constraints through dynamic programming. The recursive equation is given as

$$time_{s,e}(A) = \min_{\substack{j=0,\ldots,m_e \\ a_{e,j} \leq A}} (time_{s,e-1}(A - a_{e,j}) + t_{e,j})$$

Basically, the dynamic programming algorithm works as follows. When we are computing $time_{s,e}(A)$, we go through all the CIS versions of task T_e. For each CIS version $C_{e,j}$ that requires an area not more than A, we pre-allocate the required area and put the rest of the tasks T_{s+1} to T_{e-1} in the remaining area $A - a_{e,j}$. The execution time for this allocation is computed as $t_{e,j} + time_{s,e-1}(A - a_{e,j})$. We then choose the CIS version of task T_e with minimal resulting execution time value and record it as $time_{s,e}(A)$.

We now know how to compute the minimal execution time for the task sequence $T_{s+1} \ldots T_e$ under various area constraints. For each task sequence, the algorithm increases the area budget at every iteration, and the execution time decreases correspondingly. Hence, the area budget of the very first iteration where the execution time falls below the period constraint defines the minimal area. The constant $AREA$ is set at a large value such that all the tasks can select their best possible CIS version. The complexity of the algorithm is $O(N^2 \times AREA \times m)$, where m is the average number of CIS versions per task.

We do not take into account the communication cost between the PEs. However, it is fairly straightforward to include communication cost into our framework. We simply need to add area and performance overhead of communication while computing $area_{s,e}$ in Algorithm 2.

B. Partitioning

Algorithm 3: Compute $Area_N|P$

```
for e ← 1 to N do
    Area_e|1 = area_{0,e};
for p ← 2 to N do
    for e ← 1 to N do
        Area_e|p = min_{k=1,...,e} (Area_k|(p − 1) + area_{k,e} + areaPE)
```

Now we focus on partitioning the tasks. We define $Area_N|P$ as the minimal area required to execute tasks $T_1, \ldots, T_N$ on P processing elements such that the period constraint is not violated. Again we employ dynamic programming algorithm to compute this value. Clearly, $\min_{p=1,\ldots,N} Area_N|p$ denotes the minimal area required to execute the entire task sequence $T_1, \ldots, T_N$ on at most N processing elements.

Algorithm 3 returns the values of $Area_N|P$. The algorithm iterates over the number of processing elements p. Given a fixed number of processing elements p, we iterate over the number of tasks e. Note that $Area_e|p$ computes the minimal area required to execute tasks $T_1, \ldots, T_e$ on p PEs such that the period constraint is not violated. We need to create p partitions such that each partition will be mapped to one PE. The recursive equation is defined as

$$Area_e|p = \min_{k=1,\ldots,e} (Area_k|(p-1) + area_{k,e}) + areaPE$$

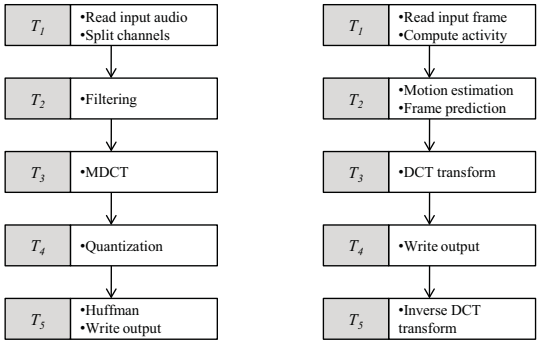
When there is only one PE, all tasks are simply mapped to it, which is the initialization statement for $Area_e|1$. The basic

idea of the recursive step is to check all possible partition points for the last PE. A partition point k partitions the task chain into two parts: task subsequence $[T_1, \ldots, T_k]$ and task subsequence $[T_{k+1}, \ldots, T_e]$. The second task subsequence $[T_{k+1}, \ldots, T_e]$ is mapped to the last PE and the first task subsequence is mapped to $p-1$ processing elements. In that case, the minimal area requirement for the last PE will be $area_{k,e} + areaPE$ where $area_{k,e}$ is the area corresponding to CIS versions computed using Algorithm 2. As we are computing our solutions iteratively, we have already computed $Area_k|(p-1)$ which corresponds to the minimal area solution for the first task sequence on $p-1$ PEs. The summation of the two returns the minimal area with last partitioning point at T_k. Among all the partitioning points ($k = 1, \ldots, e$), we select the one with the minimal area requirement.

Notice that when $k = e$, the second task subsequence will be empty and $area_{e,e} = 0$. This will essentially create additional idle PE in the end, which will increase the area by $areaPE$ without any performance benefit. Hence this solution will be eliminated. Similarly, if $e < p$, that is, the number of tasks is less than the number of PEs, we will also get some idle PEs. These idle PEs will add to area without contributing to performance. Again these partial solutions with idle PEs will not be part of the optimal solution.

The complexity of Algorithm 3 is $O(N^3)$.

VII. Experiment Evaluation



a) Tasks of MP3 b) Tasks of MPEG

Fig. 1. Task graphs of MP3 encoder and MPEG-2 encoder.

For the experiment evaluation, we use two popular streaming applications, an MP3 encoder and an MPEG-2 encoder. As shown in Figure 1, each application consists of a number of tasks, which are the compute-intensive kernels.

The base processing elements used in our experiments are the extensible Tensilica Xtensa LX2 processor cores that can be configured for applications-specific instruction set extension. Together with a hardware multiplier, 32KB of data caches, and 4KB of instruction cache, each Xtensa LX2 processor requires about 231K gates, and can run at 326MHz using 0.13μm LV manufacturing process.

For each task, we use XPRES compiler provided by Tensilica to generate a number of different configurations with

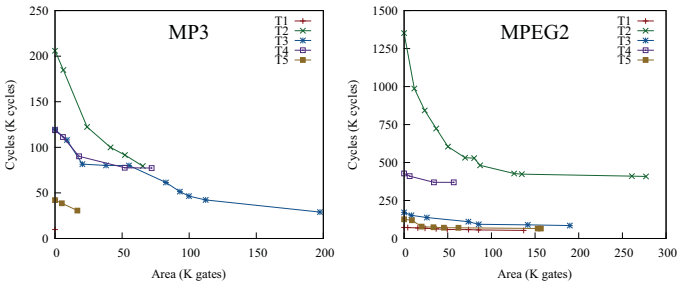


Fig. 2. Custom instruction sets for the tasks in MP3 and MPEG-2

varying trade-offs between area and performance. The CIS versions for each task are shown in Figure 2. The X-axis represents the area (in gates) and the Y-axis represents the execution time of the task. Some of the CIS versions require almost the same area as the base PE.

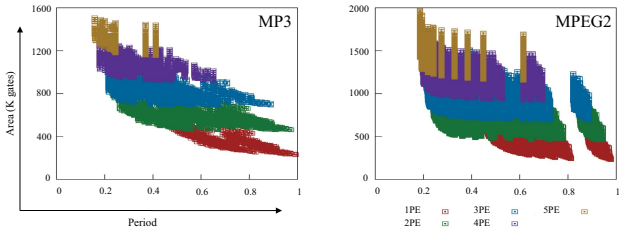


Fig. 3. Design space for MP3 encoder and MPEG-2 encoder.

We first plot the result of the exhaustive design space exploration shown in Figure 3. There are 14,400 points in the MP3 encoder design space and 387,072 points in the MPEG-2 encoder design space. The X-axis represents the period normalized with respect to the completely software based implementation on a single PE. The Y-axis represents the total area required by the MPSoC solution. Each color corresponds to the number of PEs in the solution. As can be seen from the figure, the design space is quite complex. It is possible to meet the same period constraint either with a small number of PEs each customized heavily or with a larger number of PEs devoid of customization.

Now we focus on generating the area-optimal solution under a given period constraint. For each application, we vary the period constraints from 0 to 1.0 (in steps of 0.01) of the period with pure software implementation on a single PE without any customization. The software execution on a single PE without customization is the solution with minimum area. For clarity, we plot the results for different number of PEs though our algorithm can easily identify the optimal number of PEs.

Figure 4 plots the results for the two applications. The light blue region in the left of each graph corresponds to the infeasible region where the period constraint is too small. The white region under the curves corresponds to the infeasible design space due to tight area budget. The third region, in light green, is the feasible design space. The Pareto-optimal solutions in this feasible design space are highlighted in the figure. Given a period constraint, the corresponding optimal point tells us how many PEs should be used and the minimal

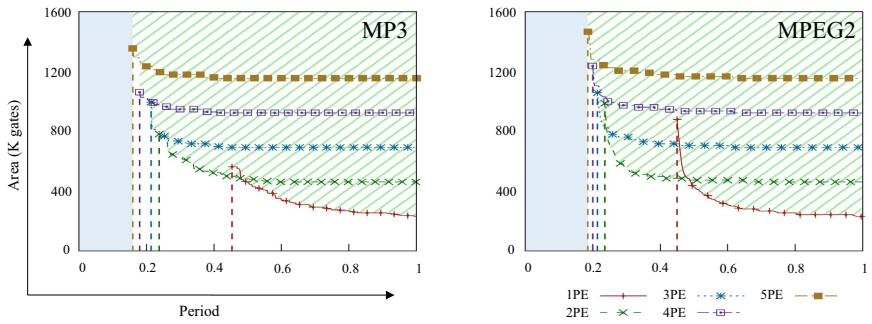


Fig. 4. Minimal area cost versus period constraint for MP3 and MPEG-2 for different numbers of PEs

Number of tasks	EA	ILP	DP
5	0.01 sec	1 sec	0.01 sec
7	1.18 sec	5 min	0.01 sec
10	12 min	9 min	0.03 sec
12	18 hour	2 hour	0.05 sec
15	-	-	0.09 sec
20	-	-	0.20 sec

TABLE 1
ANALYSIS TIME FOR EXHAUSTIVE (EA), ILP, AND THE DYNAMIC PROGRAMMING (DP) ARPPROACH.

area cost. The vertical dashed lines indicate the maximum accelerations that can be gained for different numbers of PEs.

Finally we compare the analysis time for exhaustive algorithm (EA), ILP solver and our proposed dynamic programming algorithm (DP) on Intel Xeon 2.53GHz processor with 16GB memory. We used LINGO, a commercial ILP solver [14] for our experimental evaluation. For this set of experiments, we generate synthetic task graphs with number of tasks varying from 5 to 20. The average number of CIS version per task is set at 5. The performance gain of each CIS version ranges between 1,000 to 10,000 time units. The hardware area is between 1 to 100 units.

Table 1 shows the analysis time for the three methods. The analysis time corresponds to finding the area-optimal solutions given a fixed period constraint. Given an application and a fixed period constraint, the analysis time remains unchanged for different runs of exhaustive algorithm and dynamic programming approach. However, for the ILP solver, analysis time can vary; so we report the average analysis time.

As shown in the table, dynamic programming approach improves the analysis time dramatically and still produces the optimal solution. With 15 tasks and more, exhaustive algorithm and ILP solver fail to return optimal solutions within a reasonable time. However, dynamic programming approach still manages to identify the optimal solution within short time.

The exhaustive algorithm is more powerful than ILP solver if the designer is interested in all the Pareto-optimal solutions, that is, the tradeoff between area and period. The exhaustive algorithm can explore the entire design space in one go. The ILP solver, on the other hand, needs to be invoked with different period constraints. Even the dynamic programming approach needs to be invoked with different period constraints. However, our experiments show that dynamic programming approach is way faster than exhaustive algorithm for a task graph with 12 tasks and 100 different period constraints.

VIII. CONCLUSION

In this paper, we propose an efficient hierarchical algorithm to design the most resource-efficient customized MPSoC platform for mapping linear task graphs representing streaming applications under deadline constraints. Using two popular streaming applications (MP3 encoder and MPEG-2 encoder) with Tensilica extensible processors, the experimental validation confirms the efficiency of our approach.

Acknowledgements. This work was partially supported by MOE Singapore research grant MOE2009-T2-1-033.

REFERENCES

[1] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, 2000.
[2] H. Javaid and S., "Parameswaran. Synthe sis of heterogeneous pipelined multiprocessor systems using ILP: JPEG case study," in *CODES+ISSS*, 2008.
[3] S. Shee and S. Parameswaran, "Design methodology for pipelined heterogeneous multiprocessor system," in *DAC*, 2007.
[4] A. Tumeo *et al.*, "Mapping pipelined applications onto heterogeneous embedded systems: a bayesian optimization algorithm based approach," in *CODES+ISSS*, 2009.
[5] F. Sun *et al.*, "Application-specific heterogeneous multiprocessor synthesis using extensible processors," *IEEE TCAD*, vol. 25, no. 9, 2006.
[6] Tensilica Inc., "XTensa LX2 Embedded Processor Core," http://www.tensilica.com.
[7] A. Benoit and Y. Robert., "Mapping pipeline skeletons onto heterogeneous platforms," *Journal of Parallel and Distributed Computing*, vol. 68, no. 6, 2008.
[8] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, 2004.
[9] P. Hansen and K.-W. Lih, "Improved Algorithms for Partitioning Problems in Parallel, Pipelined, and Distributed Computing," *IEEE Transactions on Computers*, vol. 41, no. 6, 1992.
[10] M. A. Iqbal and S. H. Bokhari, "Efficient Algorithms for a Class of Partitioning Problems," *IEEE Transactions Parallel and Distributed Systems*, vol. 6, no. 2, 1995.
[11] D. M. Nicol and D. R. O'Hallaron, "Improved Algorithms for Mapping Pipelined and Parallel Computations," *IEEE Transactions on Computers*, vol. 40, no. 3, 1991.
[12] A. Gerasoulis and T. Yang, "A comparison of Clustering Heuristics for Scheduling Directed Acyclic Grahphs on Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, 1992.
[13] J. Cong, G. Han, and W. Jiang, "Synthesis of an Application-Specific Soft Multiprocessor System," in *FPGA*, 2007.
[14] Lindo System Inc., "Lingo," http://www.lindo.com.