Master's Thesis of Science

SDF/L Graph Scheduling onto Heterogeneous Multicore Processors

이종 멀티 코어 프로세서에서 SDF/L 그래프 스케줄링 기법

AUGUST 2021

Seoul National University
Graduate School of Engineering
Department of Computer Science and Engineering

Mari-Liis Oldja

SDF/L Graph Scheduling onto Heterogeneous Multicore Processors

이종 멀티 코어 프로세서에서 SDF/L 그래프 스케줄링 기법

Supervisor Soonhoi Ha

Submitting a Master's Thesis of Computer Science and Engineering Studies

June 2021

Graduate School of Engineering

Seoul National University

Department of Computer Science and Engineering

Mari-Liis Oldja

Confirming the Master's Thesis written by Mari-Liis Oldja

July 2021

Chair	Prof. Jihong Kim
Vice Chair	Prof. Soonhoi Ha
Examiner	Prof. Chang-gun Lee

Abstract

Although dataflow models are known to thrive at exploiting task-level parallelism of an application, it is difficult to exploit the parallelism of data. Data-level parallelism can be represented well with loop structures, but these structures are not explicitly specified in most existing dataflow models. SDF/L model was introduced to overcome this shortcoming by specifying the loop structures explicitly in a hierarchical fashion. To the best of our knowledge however, scheduling of SDF/L graph onto heterogeneous processors has not been considered in any previous work.

In this dissertation, we introduce a scheduling technique of an application represented by the SDF/L model onto heterogeneous processors. In the proposed method, we explore the mapping of tasks using an evolutionary meta-heuristic and schedule hierarchically in a bottom-up fashion, creating parallel loop schedules at lower levels first and then re-using them when constructing the schedule at a higher level. To verify the efficiency of the proposed scheduling methodology, we apply it to benchmark examples and randomly generated SDF/L graphs.

Keywords : Mapping and Scheduling, Hierarchical Scheduling, Data-parallel Scheduling, Data flow architectures, Design Space Exploration

Student Number : 2019-24125

The author of this thesis is a Global Korea Scholarship scholar sponsored by the Korean Government

Contents

Abstract . . . . . **i**

Contents . . . . . **ii**

List of Figures . . . . . **iv**

List of Tables . . . . . **v**

Chapter 1 Introduction **1**

Chapter 2 Related Work **6**

2.1 SDF Scheduling with Data-level Parallelism 8

2.2 Hierarchical Scheduling 9

Chapter 3 Problem and Challenges **11**

3.1 Notations and Problem Description 11

3.2 Challenges 12

Chapter 4 Proposed methodology **15**

4.1 Mapping Exploration 15

4.2 Priority Assignment and List Scheduling Heuristic 17

4.3 Hierarchical Scheduling 18

4.4 Complexity 23

Chapter 5 Experiments **24**

5.1 Benchmarks 25

5.2 Randomly Generated Graphs 30

Chapter 6 Conclusions **35**

Bibliography **37**

요 약 **41**

List of Figures

Figure 1.1 An example of hierarchical SDF/L graph 3

Figure 1.2 Overall structure of the proposed methodology 5

Figure 3.1 Two scheduling options for the lower-level graph of Fig. 1.1: (a) scheduling on a single processor and extending it horizontally and vertically and (b) Scheduling on two processors and extending it horizontally. Colors indicate different iterations. 12

Figure 3.2 Two scheduling options for the middle-level graph of Fig. 1.1: (a) L2 is mapped to both processors, (b) L2 is mapped to $P1$ only . . . 13

Figure 3.3 Example chromosomes 14

Figure 4.1 Priority assignment based on the priorities of depending instances 18

Figure 4.2 Scheduling examples of a D-type loop node 20

Figure 4.3 Scheduling example of a loop node 22

Figure 5.1 The example of hierarchical SDF/L graph after flattening 25

Figure 5.2 SDF/L benchmarks 26

Figure 5.3 Pareto-optimal solutions for three benchmarks. 29

Figure 5.4 Comparison of average iteration time 31

Figure 5.5 Comparison of latency 31

Figure 5.6 Randomly generated graphs schedule latency and iteration time comparisons 31

Figure 5.7 Iteration time improvement over the loop count increase for two random graphs 33

List of Tables

Table 5.1 Execution time information of the synthetic example 27

Table 5.2 Design space exploration results 27

Table 5.3 Improvement over the flat scheduling approach for randomly generated SDF/L graphs 31

Chapter 1

Introduction

Heterogeneous multiprocessor embedded systems with multi-core CPUs and GPUs have emerged to support computation-intensive applications. Such applications often include data-parallel computations, procedures that we need to process in a quick and parallel manner on the embedded system. Deep learning applications are a great example of such applications, to run inference fast, it is critical to distribute the data-parallel computations onto different processing units for efficiency.

Dataflow models have been widely used for specifying embedded system applications given the inherited task-level parallelism in the model itself. Particularly, Synchronous Dataflow (SDF) model [1] is often used for streaming applications due to its formality and predictability. The model displays an application as a graph where each node indicates a functional task, and directed edges represent channels to transmit data (sample)[1] between those functional tasks.

Each node consumes samples to be executed and produces samples needed by the other tasks as a result of the execution. Based on the sample rates fixed in advance, we can easily determine the repetition count for each node in a periodic schedule of the graph and using this information, it is possible to decide the static schedule of the application at compile time. The lowest graph in Fig. 1.1 serves as an example of the SDF graph

[1]In this paper, we interchangeably use node and task in the same meaning. The task instance, or job, is the unit of schedule.

representation of an application. Node *A2* produces a single sample to node *Z* when executed, node *Z* however needs to consume 2 samples to execute in turn. This means that for node *Z* to run, node *A* should run twice to produce enough samples. Nodes *A* and *B* have a one to one connection between them meaning, for every execution of node *A*, node *B* will also be executed, this makes the repetition count of node *B* also two.

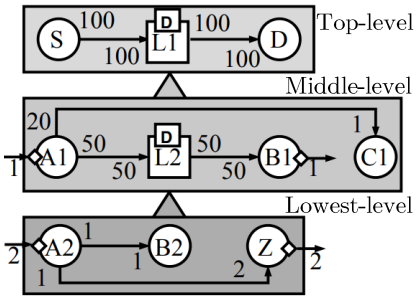
The SDF graph is often presented as a hierarchical graph, where a node may be representing another graph, in which case the node is referred to as a *macro* node.

As an extension, Synchronous Dataflow with Loop (SDF/L) model [2] was recently proposed in order to, without the loss of SDF property, express the computation-intensive and data-parallel applications which contain loops. The SDF/L graph similarly has a hierarchical structure, adding a special type of macro node that represents a loop structure. The biggest difference from the original model is that the SDF/L model explicitly shows not only the task-level parallelism but also expresses the data-level parallelism through the use of these loop nodes.

In the SDF/L model, two types of loops can be described - data loop (D-type) and convergent loop (C-type). The D-type loop is used to explicitly express data-level parallelism. If a macro node represents a D-type loop, iterations of the subgraph can be run in parallel on multiple mapped processors. Meanwhile, a C-type loop is used to express a loop with the conditional exit similarly to *break* keyword commonly used in programming languages. Running C-type loop iterations concurrently is not allowed as it is necessary to check the exit condition at the completion of each iteration.

Thanks to its extended expressiveness, the SDF/L model can be used to describe deep learning applications as dataflow graphs which previously was difficult due to the lack of data-parallel structures. These macro nodes representing the repeating loop tasks, that can be consecutively executed on multiple processors, allow us to divide the graph into multiple levels and provide the hierarchy we wish to exploit in this paper.

Figure 1.1 shows an example of an SDF/L graph. The diamond port on a node in the



Level (# of iteration)	Task	Rep. count in each level
Top (1)	S, L1, D	1
Middle (100)	A1, B1, L2	1
	C1	20
Lowest (25)	A2, B2	2
	Z	1

Figure 1.1: An example of hierarchical SDF/L graph

nested graph represents a data flow to or from a higher-level graph. For example, node *A1* consumes 1 sample from node S in the top-level. Node *B1*, defined in the middle-level, produces 1 sample to top-level node D. In the top-level graph, node S executing once produces 100 samples to node $L1$, and node D requires 100 samples to run. Since each iteration of the middle-level graph produces and consumes 1 sample, with the total of 100 iterations of the middle-level graph, the required samples are produced.

Going into more detail, there are two types of ports: *broadcasting* and *distributing*. The former is used to broadcast the same data for all iterations, while the latter is used to transmit different data to each iteration. When different ports are used, the sample rates are also different, representing a queue and a buffer type input, respectively, which in a simple sense means that with the broadcasting port, the same sample can be re-used while samples coming in through distributing port are discarded after use. In the example figure, the distributing ports are used for ease of explanation.

Keep in mind though, that scheduling graphs on a multiprocessor system is an NP-hard problem [3] and finding the best schedule is not a trivial problem. Although dataflow models have this hierarchical structure, most of the previous works have focused on mapping and scheduling a flattened graph. Many methods have been proposed to find efficient mapping and scheduling on multiprocessor systems based on a flattened SDF graph.

The authors in [4] and [5] applied the mathematical approach using integer linear programming (ILP). Also commonly used in an attempt to solve this problem are approaches using meta-heuristics - ([6, 7, 8, 9, 10, 11, 12, 13]). There is a number of scheduling heuristics that have been proposed, some examples are [14] and [15].

However, given that a graph with nested loops can get exponentially big when flattened, these approaches making use of the flattened graph to compute the schedule are not best suited for the exploration of mapping and scheduling of an SDF/L graph.

In addition, if we get rid of the structure of the graph, the information of the composition of the loop is ignored, and the loop scheduling containing repeating tasks gets mixed with normal tasks - this can make code synthesizing difficult.

To solve those problems, [14], [16] and [17] proposed hierarchical scheduling on the SDF graph to make the schedule much simpler without too significant loss of accuracy. However, none of them consider heterogeneous processor setups, which are prevalent for an embedded system. To the best of our knowledge, there is no previous work that considers scheduling of SDF/L graph onto heterogeneous processors.

In this paper, we propose a mapping and scheduling methodology for an application described with the SDF/L graph onto heterogeneous multiprocessors with a goal to minimize the latency of the graph and number of used processors in a feasible time, while taking advantage of the structure of the specific graph.

Figure 1.2 shows the overall structure of the proposed scheme, where we separate mapping and scheduling. We use a meta-heuristic algorithm to decide the mapping of tasks onto processors. With a given mapping, we use heuristics for priority assignments and composing the schedule.

The main contribution of this work lies in the heuristic to schedule the graph hierarchically in a bottom-up fashion, given an SDF/L graph divided into levels through the explicit loop structure.

The scheduling objective is to minimize the latency in a much shorter time than

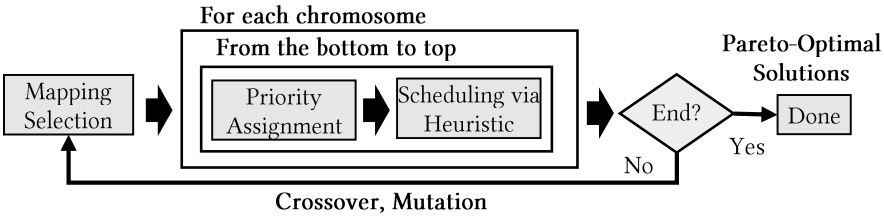


Figure 1.2: Overall structure of the proposed methodology

scheduling based on a flattened graph with minimal loss of accuracy. Additional objective in the meta-heuristic algorithm is to minimize the number of processors used in hopes to utilize processors more effectively.

Chapter 2

Related Work

Numerous scheduling techniques have been proposed for SDF graphs [18]. They can be classified into three approaches: heuristic, meta-heuristic, and integer linear programming(ILP). We review the related work focusing on the scheduling of SDF graphs onto heterogeneous processors first.

The authors of [19] proposed a list scheduling heuristic for the mapping and priority assignment of tasks in DAG called Best-Imaginary-Level (BIL) scheduling. The node priority is determined by the critical path length from a node considering heterogeneous processors. It is proven that it generates an optimal schedule for the graph with a linear topology. In the work of [20], Heterogeneous Earliest-Finish-Time (HEFT) is presented, which is similar to the BIL scheduling.

The authors of [21] proposed another list scheduling heuristic for real-time applications that have deadline constraints. The node priority is computed by the rest time until the deadline, called laxity.

Although heuristic-based methods are fast and provide near-optimal results, they are tailored for the given problem setting and are not expandable to reflect additional constraints or extended graphs. To apply those techniques to the SDF graph, the conversion of SDF to DAG is needed.

The genetic algorithm (GA) is a popular meta-heuristic that is used for mapping and

scheduling problems of SDF graphs thanks to its extensibility and availability of solution frameworks. The authors of [6] investigated the effects of multi-objective evolutionary algorithms (MOEAs) for the mapping problem of heterogeneous MPSoC design. An application is assumed to be specified by a Kahn process network [22] and three objective functions are considered: processing time, power consumption, and cost. They conducted comparative experiments extensively, changing algorithms and strategies.

The work in [9] proposed a two-phase mapping methodology for mapping and scheduling multiple HSDF graphs. In the first phase, they generate a set of schedules for each application on a varying number of processors by applying the genetic algorithm. In the second phase, those schedules are merged based on the schedulability analysis of each processor.

The authors of [10] presented a mapping and scheduling methodology of an application represented by a DAG through the genetic algorithm. They showed the trade-off relation among lifetime reliability, power consumption, and maximum execution time on the heterogeneous platforms.

The work in [11] presented a genetic algorithm-based technique for mapping multiple SDF graphs onto heterogeneous processors, proposing a clustering technique to reduce the model conversion overhead from SDF to DAG.

The author of [12] proposed the mapping scheme of an MMDF (multi-mode dataflow) graph on heterogeneous multiprocessors. They proposed a cooperative co-evolutionary genetic algorithm (CCGA) which efficiently explores the design space by a problem-specific decomposition strategy in which the node mapping solutions for each individual mode are assigned to an individual population.

The author of [13] considered single or multiple deep learning applications on a heterogeneous system that includes both GPU and NPU (Neural Processing Unit), exploiting the task-level parallelism of each network.

ILP (integer linear programming) is popularly used to find an optimal solution for

small-scale mapping and scheduling problems. The work in [23] produced an optimal solution via an ILP-based method considering task mapping/scheduling as well as SPM partitioning. The authors of [24] presented an ILP-based framework to map an application represented by a graph to heterogeneous chip multiprocessors within the energy and reliability constraints. The authors of [25] also proposed an ILP-based method, aiming to minimize the latency of the DAG schedule on heterogeneous processors, considering bus contention. The authors of [26] solved the mapping and scheduling problem of the acyclic SDF graph with a satisfiability modulo theory (SMT) solver.

2.1 SDF Scheduling with Data-level Parallelism

Even though SDF scheduling has been extensively researched, there is relatively small number of works considering data-parallelism. The authors in [15] presented a heuristic algorithm for partitioning an SDF graph on homogeneous multiprocessors to optimize the arrival rate. They find hot actors which are stateless and a bottleneck of the program, and duplicate them. To resolve the bottleneck, they exploit data-level parallelism by allocating the duplicated actors on different processors. The work in [27] introduced a compiler for Streamlt programming language [28] that is based on SDF. To leverage the data-level parallelism, the stateless actors are detected, and undergo the fission to map to distinct cores.

An ILP-based approach was proposed to scheduling SDF graph onto heterogeneous architecture [5], exploiting not only task-level parallelism but also data-level parallelism. The authors in [4] also presented an ILP-based technique taking into account the data-level parallelism, and additionally performed granularity-based optimization on heterogeneous architectures. Due to the limitation of the ILP, however, neither method is suitable for handling a large number of nodes as the time complexity of the ILP solver is exponential. The same authors of [5] proposed a scheduling technique on heterogeneous processors using Quantum-inspired Evolutionary Algorithm (QEA), which finds a near-optimal solution

much faster than their previous work [7].

The authors of [29] converted a convolutional neural network (CNN) to an SDF graph and made the mapping decision through the genetic algorithm by exploiting both task-level and data-level parallelism. Then, they converted the CNN to a cyclo-static dataflow (CSDF) graph based on the mapping to represent the final platform-aware executable CNN inference application.

All these works schedule the flattened graph and consider only a single task for data-parallel execution. On the other hand, we consider data-parallel execution of the full subgraph inside a loop.

2.2 Hierarchical Scheduling

The SDF graph may have many instances of nodes because of the sample rate mismatch, which increases the scheduling complexity. To reduce the scheduling complexity, the authors of [14] proposed a hierarchical scheduling heuristic on homogeneous multiprocessors. They first cluster nodes, then combine clusters hierarchically to reduce the makespan of an SDF graph.

The work in [16] proposed a hierarchical compilation method for a macro dataflow graph. They used a tree-based heuristic to decide the number of processors on nodes in a macro node, then schedule nodes hierarchically. In their method, however, the number of processors is decided without considering the scheduling, and it may cause an empty hole in the schedule leading to an inefficient schedule. The most closely related work to the proposed technique is [17] where quasi-static scheduling of dynamic constructs is performed hierarchically. They generated the schedule of one iteration in the macro node, and simplified the macro node as a node which has a repeated pattern of one iteration. Thanks to the hierarchical nature of the approach, the scheduling method scales to relieve the complexity with a decent result compared to the ideal schedule. However, none of the above schemes support heterogeneous processors.

The authors of [30] proposed a hierarchical scheduling technique on DAG. They assume the hierarchical task graph (HTG) which is a nested DAG. They recursively schedule using ILP from subgraph in a bottom-up manner. Since ILP is time-consuming and it is repeated many times, this method is not scalable.

Chapter 3

Problem and Challenges

3.1 Notations and Problem Description

The purpose of this work is to in a reasonable time find a mapping and corresponding schedule in a hierarchical fashion, aiming to minimize the latency of an application specified by an SDF/L graph while using minimal number of processors.

An SDF graph g is characterized by a tuple (N_g, E_g), where N_g and E_g represent the set of nodes and the set of edges, respectively. $|N_g|$ indicates the number of nodes. The repetition count of node $n \in N_g$ is denoted by $Rep(n)$. The number of incoming/outgoing edges of node n is represented as $|E_n^{in}|$ and $|E_n^{out}|$, respectively. Instances of a node n are represented as $\{n_i\ (1 \leq i \leq Rep(n))\}$, each of which is assigned a different priority $Pr(n_i)$. We indicate a set of node instances as $\dot{N}_g$. Furthermore, we define $Dep(n_i)$ as a set of depending instances of instance n_i, and $Pr(Dep(n_i))$ as a set of priorities of the depending instances of n_i.

The application specified above is executed on a set of heterogeneous processors denoted as $\mathcal{PE}$. The number of processors is represented as $|\mathcal{PE}|$. Each node n has an execution time $C(n, k)$ on the mapped processor $P_k \in \mathcal{PE}$. If the mapped processor of a node is known, we simply use $C(n)$ to indicate the execution time of the node. To make the problem practical, we consider the constraint where some node can not be mapped to a certain processor by setting the execution time to infinity. For instance, CPU tasks may

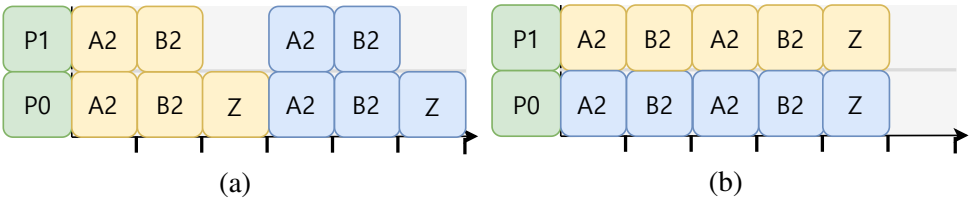


Figure 3.1: Two scheduling options for the lower-level graph of Fig. 1.1: (a) scheduling on a single processor and extending it horizontally and vertically and (b) Scheduling on two processors and extending it horizontally. Colors indicate different iterations.

not run on GPU due to the algorithm of the task, or the user may fix the mapping of a data-parallel task to GPU for efficient processing.

We assume that the nodes inside a D-type loop are mapped onto a set of homogeneous processors for data parallel execution, but the user can specify some specific heterogeneous mapping for any node even if it is contained within the loop. For example, in Fig. 1.1, while macro node *L2* generally can be mapped to either CPU or GPU, not both, the user can enforce node *Z* to be mapped to GPU even when *L2* as a unit is mapped to CPU. In the case of C-Type loop, the inside nodes are not data-parallel as previously mentioned and so there is no unit mapping, nodes can be mapped to heterogeneous processors. For the latency evaluation of the mapping, we assume the fixed loop iteration for C-Type loop as the maximum iteration count allowed for conservative and safe scheduling considering the worst case.

In summary, the scheduling problem tackled in this work is how to map and schedule an SDF/L graph *hierarchically* onto heterogeneous processors to minimize the latency with given profiling information of task execution times on the processing elements and considering a given set of mapping constraints.

3.2 Challenges

Before going into details about the proposed approach, we need to explain some challenges involved in hierarchical scheduling. Since a loop is executed repeatedly, the

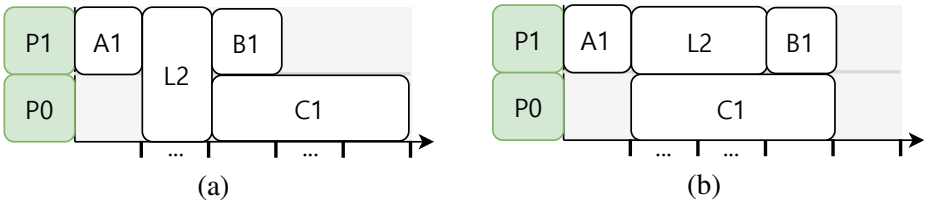
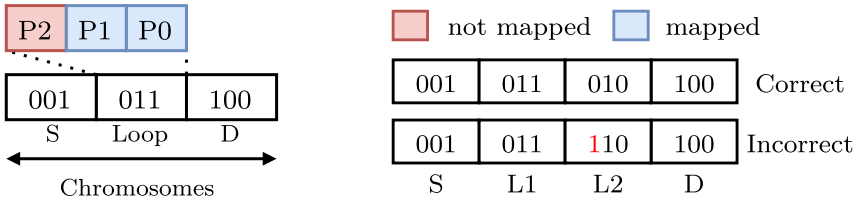


Figure 3.2: Two scheduling options for the middle-level graph of Fig. 1.1: (a) L2 is mapped to both processors, (b) L2 is mapped to $P1$ only

first challenge is to find a throughput-optimal schedule of a loop node for a given number of processors. In the case of D-type loop that can be executed in parallel, we have several options for how many processors to use to schedule a single iteration. Figure 3.1 shows two scheduling options for the lower-level graph of Fig. 1.1. If we schedule an iteration on a single processor and extend the schedule horizontally and vertically, we may end up with an idle period due to the unbalanced assignment of loop iterations between processors (Fig. 3.1(a)). If scheduling of an iteration is done with two processors and extended horizontally, we may have an idle period due to unbalanced workload assignment at each iteration, (Fig. 3.1(b)). We need to consider both cases in finding the schedule of a data parallel loop. While we may use any existing technique to find a throughput-optimal schedule of an SDF graph for a given number of processors, we devise a list scheduling heuristic to find a sub-optimal schedule fast.

The second challenge comes from the nature of hierarchical scheduling. Since we schedule hierarchically from the lowest level nested graph, catching the information of surrounding nodes is more difficult than it is for flattened scheduling. If the mapping of nodes in the inside loop overlaps with the mapping of outside nodes, the contention may occur on the same processor which makes it hard to schedule efficiently. Figure 3.2(a) and 3.2(b) show this difficulty explicitly. There is a nested SDF/L graph as shown in the middle- and lower-level graph in Fig. 1.1 for an easy explanation. Since we schedule hierarchically, it is hard to consider other nodes in the upper level graph when we schedule the inside of *L2*. If *L2* is scheduled as shown in Fig. 3.2(a) and node *C1* has a very long



(a) An example chromosome (b) An example chromosome if loop *L2* is nested in loop *L1*

Figure 3.3: Example chromosomes

execution time, the resulting schedule is not efficient. Even though the latency of *L2* in Fig. 3.2(b) is higher than for the one shown in Fig. 3.2(a), it may result in a better higher level schedule. To tackle this challenge, we separate mapping and scheduling, exploring the mapping of nodes using an evolutionary meta-heuristic.

Chapter 4

Proposed methodology

The proposed methodology consists of three steps described in Fig. 1.2. First, we generate the mapping of nodes using a meta-heuristic algorithm. In the next step, we assign priorities and compose the schedule using the heuristics with the generated mappings. We schedule hierarchically from the bottom up and make an evaluation based on the scheduling results. By exploration via a meta-heuristic algorithm, we eventually find Pareto-optimal mapping and scheduling results in terms of latency and the number of processors.

4.1 Mapping Exploration

We make use of a genetic algorithm (GA) as a meta-heuristic for the exploration of mapping. Any meta-heuristic can be applied, but GA is chosen because of a well-established GA solver [31] being freely available. During this process, we first generate many chromosomes that represent candidate mapping solutions and calculate each chromosome's fitness value which based on the latency of the schedule produced when using the mapping and the number of processors included in the mapping. Among chromosomes, we select some chromosomes which have good fitness values and apply GA operations such as mutation and crossover to produce the offspring chromosomes. The offspring then replace the existing chromosomes with poor fitness values. This procedure

is repeated until the fitness value is converged or the predefined maximum number of iterations is reached.

Since the nodes inside a D-Type loop may be assigned to more than one processor, the chromosome allowing *multi-mapping* of a node to multiple processors is devised. Figure 3.3 shows an example chromosome. For easy explanation, we assume that there are three processing elements *P0*, *P1*, and *P2* in $\mathcal{PE}$. In the figure, each gene in the chromosome represents a mapping of a node in the top-level graph or a macro node. A gene consists of a bit array, each of which expresses whether the corresponding processor is included in the mapping of the said node or not. For example, the *Loop* node chromosome in Fig. 3.3(a) indicates that the nodes inside the *Loop* macro node are to be scheduled on processors *P0* and *P1*. Note that no gene is assigned for the normal nodes inside a D-type loop since we assume that a D-type loop is assigned to homogeneous processors only in order to exploit the data-level parallelism.

We could consider another option of mapping all nodes inside the D-type loop explicitly. Then the scheduling and performance evaluation would be easy but the design space of mapping may become too wide to explore efficiently. Hence we specify the mapping of the D-type loop as a whole and resort to a scheduling heuristic to determine the mapping and scheduling of nodes inside a D-type loop, which reduces the design space of mapping drastically. For a C-type loop, however, we specify the mapping of each inside node individually since the loop can not be executed in parallel and the nodes can be mapped to heterogeneous processors.

If some node is manually mapped to some specific processor, no gene associated with the node is added to the chromosome since there is no need to explore the mapping for such nodes. If there is a nested D-type node inside a loop such as shown in Fig. 1.1, then we also add a separate gene for the nested macro node as displayed in Fig. 3.3(b). In this case, there is a constraint that node *L2* should be mapped to a subset of the processors mapped to the parent loop at the higher level *L1*. During the GA operations, an incorrect

chromosome may be generated such as the lower chromosome shown on Fig. 3.3(b). To handle this and other cases of unwanted or incorrect chromosomes, a repair process is induced to partially re-generate the offending part of the chromosome.

To sum up, a chromosome includes the mappings for normal top-level nodes and the mapping of each loop node regardless of the level.

4.2 Priority Assignment and List Scheduling Heuristic

List scheduling is a popular approach used for scheduling, especially in multiprocessor systems with parallelizable tasks. We use list scheduling as a base and introduce a number of small improvements compatible with our approach to speed up the scheduling process. For list scheduling, we use a priority assignment method based on Latest Starting Time (LST) which is similar to [19]. The proposed method allows us to do priority assignment calculations in a more compact way using the SDF graph itself instead of extending the SDF graph into a homogeneous synchronous dataflow (HSDF) where each instance of an SDF node becomes a separate node, while still giving different priorities to different instances of the same task.

To do priority assignment, we first initialize nodes with priority arrays of length corresponding to the repetition count, the values in the array itself are initialized by multiplying the instance index and the execution time of each node. For example, in the case of node *B2* in Fig. 4.1 which is in the lowest graph of Fig. 1.1, $Pr(B2_1) = 1 \cdot 1 = 1$ and $Pr(B2_2) = 2 \cdot 1 = 2$. Note that as we have fixed the mapping at this point, the execution time for each node is fixed.

We consider instances to be connected if one produces data samples for the other. Knowing the repetition counts of two connected nodes we can easily calculate which instance of the producing node produces samples for which instance of the consuming node using equation (4.1). The Id_S indicates the instance id of the source node *S*, and Id_D is the instance id of the destination node *D* corresponding to the instance S_{Id_S}.

$$Id_D = \lfloor \frac{Id_S * Rep(D) - 1}{Rep(S)} \rfloor + 1 \tag{4.1}$$

$$Pr(S_{Id_S}) = Max(Pr(S_{Id_S - 1}), Pr(Dep(S_{Id_S})) + C(S_{Id_S}) \tag{4.2}$$

Starting from the exit nodes and moving upwards the entry nodes, we calculate the priority of an instance by taking the maximum of the priority values among depending instances and its previous instance, and adding the execution time as described in equation (4.2). If i is zero, then $Pr_{S_{i-1}}$ is zero. For example, we calculate the Id_D for nodes $B2$ and Z to get $Pr(A2_2)$. From equation (4.2), Id_D for node $B2$ is computed to be 2, and Id_D for node Z is one. For $Pr(A2_2)$ computation, we perform Max operation among three values: $Pr(A2_1)$, $Pr(B2_2)$, and $Pr(Z_1)$ (blue and green arrows in Fig. 4.1). In this way, we assign priorities in a hierarchical manner before scheduling.

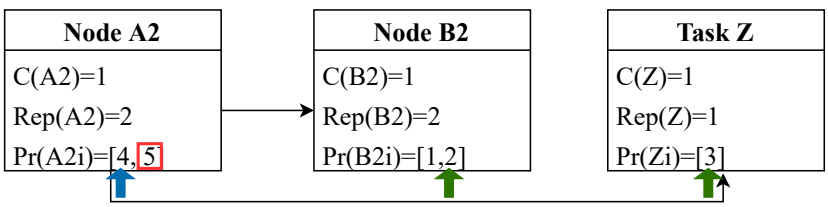


Figure 4.1: Priority assignment based on the priorities of depending instances

The complexity of priority calculation depends on the instance counts and the number of dependencies, which is $O(\sum_1^N Rep(n_i) * |E_{n_i}^{out}|)$ where N is the number of nodes. In the worst case, this could be represented as $O(|\dot{N}_g| * max(|E_i^{out}|))$ for all i.

4.3 Hierarchical Scheduling

After priority assignment is completed, we use a list scheduling heuristic to schedule the inside graphs of loop nodes hierarchically in a bottom-up fashion, starting from the lowest-level. Let us remind the assumptions made for the scheduling of a D-type loop.

The tasks inside a D-type loop have to be scheduled to a set of homogeneous processors. But there may exist some nodes within the D-type loop for which mapping to a specific processor, homogeneous or heterogeneous, is enforced by the user. For instance, the user may want to map a time-consuming node to a GPU while other nodes can be mapped to either CPU or GPU. To support this scenario, we allow mapping the D-type loop onto a multi-core CPU while mapping the designated node to GPU.

Such assumptions are made to simplify the problem while covering the common practical scenarios. If we allow the mapping of a D-type loop onto heterogeneous processors, exploitation of data-parallelism would be extremely complicated and the design space of mapping would become unbearably large.

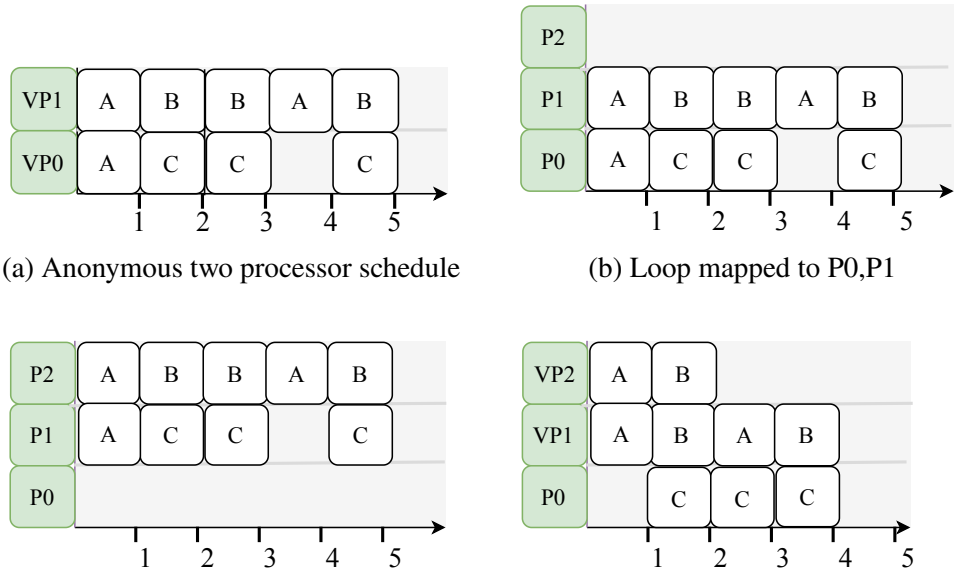
While the mapping is decided in a top down approach, the scheduling itself is done in a bottom up fashion. Since a D-type loop is mapped to homogeneous processors, different mappings can share the same schedule as illustrated with a simple example in Fig. 4.2 where we have three homogeneous processors and 2 of them are used for the loop. As shown in Fig. 4.2(a), the loop schedule should be made on a set of anonymous virtual processors (VPs) and recorded into the schedule database with a unique identifier for each set of similar mappings. If a schedule for a similar mapping is already generated, we reuse the partial schedule instead of re-calculating it. For mappings [P0, P1] (Fig 4.2(b)) and [P1, P2] (Fig 4.2(c)), we can reuse this schedule by fitting it onto the now fixed processors.

The conditions under which we consider mappings similar (to allow reuse) are as follows:

1. Mapping contains the same number of processors from the same homogeneous set of processors.

2. If there exist manually-mapped tasks within the loop, they should be mapped to the same processor.

3. If the loop mapping overlaps with the manual mapping of some tasks, the same



(a) Anonymous two processor schedule

(b) Loop mapped to P0,P1

(c) Loop mapped to P1,P2

(d) Anonymous two processor schedule with a fixed mapping of C

Figure 4.2: Scheduling examples of a D-type loop node

overlap is present in all similar mappings.

To clarify the third condition, consider the case where node C is a special task for which the mapping has been fixed to P0. In this case, the lower processor on Fig 3.1(a) would no longer be anonymous but would be fixed to P0. With C fixed, the [P0, P1](with C[P0] overlap) mapping case would be regarded as the same schedule (Fig 4.2(b)). For mapping onto [P1, P2], however, we now need to generate a different schedule as shown in Fig 4.2(d).

After starting the scheduling process for the task graph, we first check whether we have the pre-calculated schedule in the schedule database for each loop node by composing the identifier based on the mapping and looking into the database for the schedule corresponding to the composed key. If no pre-calculated schedule is present, it triggers the list scheduling heuristic for the loop node. To calculate the loop schedule, we schedule the loop one iteration at a time until we observe a repeating pattern - we find that the single iteration schedule just created overlaps with the schedule of some previous

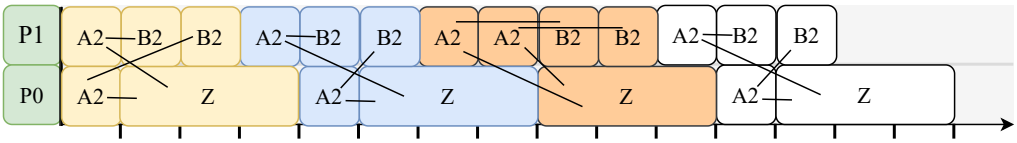
iteration. At this point we can figure out a repeating pattern and just extend the schedule to match the loop count using this pattern.

Suppose we need to schedule loop $L2$ in our example task graph shown in Fig 1.1; We have three tasks (*A2*, *B2*, and *Z*) and their execution times are 1, 1, and 3, respectively on the homogeneous set of processors considered.

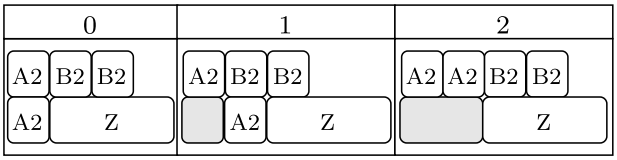
As an example, consider the case where *L2* mapping is given as *[P0, P1]*, which are two processors from a set of homogeneous processors, and task *Z* is a designated task that is manually mapped to processor *P0*. This means there is an overlap in the processors assigned for the loop and for the designated task *Z*.

How to find the repeating pattern in loop scheduling is illustrated in Fig. 4.3. As we schedule iterations of the loop step by step as illustrated in Fig. 4.3(a), we compare the schedule shape with the previously seen single iteration schedules that are shown in Fig. 4.3(b). In the figure, we distinguish the schedule of each iteration with a different color. In this example, we find the repeating pattern after the 4-th iteration. Iterations 1 and 2 make up our pattern and after identifying this repeating pattern, we can extend the schedule by adding the necessary number of appropriately shifted copies of the already found schedules to compose the full schedule of the loop. Finally, we mark the entry and exit nodes corresponding to the channels into and out of the loop and save the pre-calculated scheduling result to our schedule database to be reused for mappings with similar characteristics.

After finishing the scheduling of $L2$, we go back to scheduling our task graph at one level higher. We go on to pre-calculate the schedule for L1, after which we have enough knowledge to schedule the graph at the top level plugging in the L1 schedule where needed. After we finish scheduling a task graph containing a multi-level data-parallel loop node for a given mapping, we repeat this process for a different mapping solution (chromosome). As explained above, we reuse the pre-calculated schedule as much as possible. Note that it is possible for a higher level loop, L1, to have a non-similar mapping while a inner loop,



(a) Iteration by iteration scheduling of a loop until a repeating pattern is found



(b) Unique iteration schedules

Figure 4.3: Scheduling example of a loop node

L2, which is mapped to a subset of processors, is assigned a mapping considered similar to a previously processed mapping. Then, we need to perform scheduling at the middle level only, re-using the bottom level schedule for L2. This offers a big advantage in a GA based approach for design space exploration, considering the large number of mappings that need to get evaluated (scheduled).

Note that the schedule distribution to processors in itself (excluding the specially fixed processor) is not ordered and all permutations are valid; Whatever order we fit the schedule to the assigned processors, the mapping does not break the validity of the schedule. In an attempt to schedule the loop node in a way that the other nodes would better fit around it, we use a utilization based heuristic to decide the ordering of processors. However, the ordering will not in fact make a difference for more simple use cases, as the genetic algorithm will serve to find the best mapping of the rest of the nodes that would naturally fit around the loop schedule regardless of how we decide to order it. It will start making a difference only when we have multiple non-overlapping loops or manually mapped designated nodes.

4.4 Complexity

As the actual scheduling of a loop node is performed only once for each set of similar mappings and the schedule is reused in every subsequent same identifier mapping, the number of times we do the loop scheduling depends on the number of unique keys in our schedule database. The identifier consists of two parts. The first part depends on the number of homogeneous processors mapped to the loop, consisting of a tag for the homogeneous processor set and number of processors included. The second part indicates whether each designated task is mapped to a homogeneous processor overlapping with the other loop nodes or to a separate processor. Considering these, the maximum number of unique identifiers Ui for a single-level loop task, L, can be expressed as shown in equation (4.3) where $|\mathcal{PE}|$ represents the total number of processors in our system and $Ht(L)$ represents the number of manually-mapped tasks contained in this loop task.

$$Ui(L) = |\mathcal{PE}| * 2^{Ht(L)} \tag{4.3}$$

If we have multiple levels of loops inside a loop, the number of identifiers grows exponentially in theory. But considering that the number of manually-mapped tasks and the number of loops are usually limited in practice, the total computation time of loop scheduling is affordable, we believe. In fact, during the design space exploration using genetic algorithm, calculation of loop schedule is done for only a fraction of explored chromosomes and the time spent on pre-calculating the loop schedule turns out to be insignificant in our experiments.

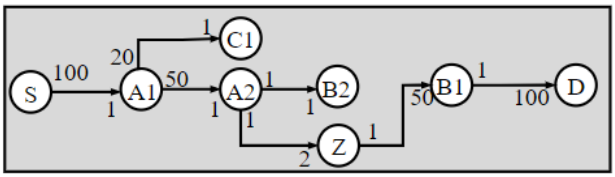
Chapter 5

Experiments

To evaluate our approach, we apply the same GA meta-heuristic to the flattened graph, which is taken as the baseline technique, called *flat* scheduling, for comparison. To flatten an SDF/L graph, we bring all of the tasks up to the top level, getting rid of the loop structure. The loop count is reflected in the repetition counts, which means that the instance number for all tasks in the final schedule is the same regardless of the scheduling approach. The flattened version of our synthetic example and the corresponding task repetition counts are presented in Fig. 5.1. The flat scheduling schedules the flattened graph as a whole without loop hierarchy. Note that the flattened graph is also an SDF graph and we do not expand the SDF graph to an HSDF graph in either of the two approaches, baseline and the proposed.

Since there are only a few SDF/L examples currently available that were introduced at the time of proposing the SDF/L model [32], we implemented and utilized a random SDF/L generator to further evaluate the effectiveness of the approach. There are two metrics considered for evaluation: scheduling time and the latency of the produced schedule.

For all experiments, we set up our exploration to consist of 100 generations with 100 chromosomes in population and 25 offspring generated in each generation with uniform crossover and 5% mutation with a rate of 95% and 70%, respectively, based on NSGA2 selector algorithm. Across the experiment results we observe the latency of the generated



Level (# of iteration)	Task	Rep. count in each level
Top (1)	S, D	1
	A1, B1	100
	C1	2000
	A2, B2	5000
	Z	2500

Figure 5.1: The example of hierarchical SDF/L graph after flattening

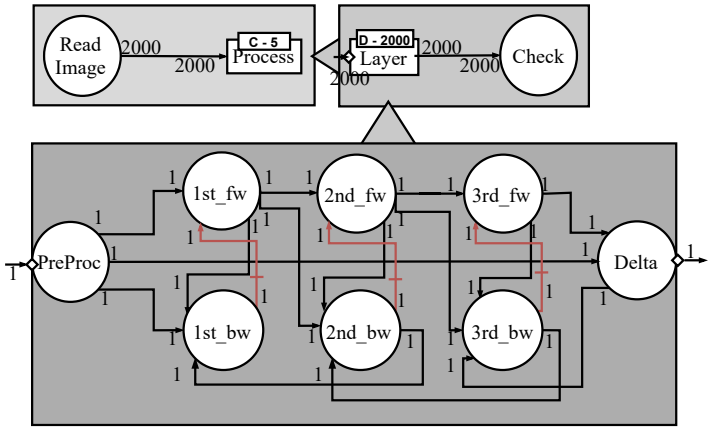
final schedule and the average iteration time of the GA based DSE algorithm. The average iteration time represents the average time it took to complete the evaluation for one generation of offspring (that is 25 mappings); It means that we perform scheduling 25 times per iteration, once for each mapping chromosome. The scheduling objectives as mentioned earlier, are to minimize the latency and the number of processors used to utilize processors more effectively.

5.1 Benchmarks

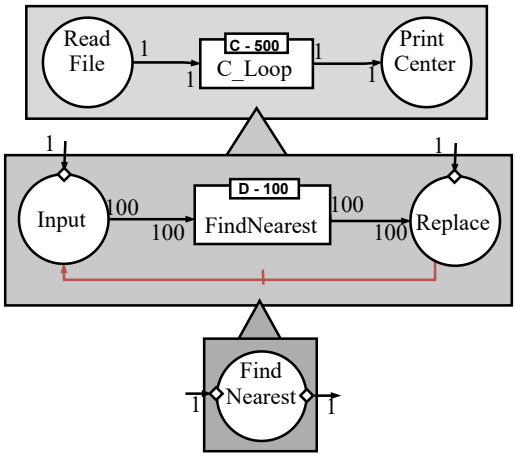
In addition to the synthetic example of Fig. 1.1, we use two benchmark programs that contain a single layer D-type loop inside a C-type loop hierarchically - *MNIST* shown in Fig. 5.2(a) and *K-means Clustering* described in Fig. 5.2(b), where the red line represents a feedback channel with initial tokens available for the first iteration.

We profiled the provided real applications on Jetson TX2 which is a heterogeneous system consisting of two Denver CPUs, four ARM A57 CPUs, and one NVIDIA Pascal GPU: the total number of PEs is seven. In the case of *MNIST*, codes for all processors are provided and profiling is performed for two types of CPUs and GPUs. In the case of K-means Clustering, however, only CPU codes are provided, so only two types of CPUs are profiled and the design space is explored. For the synthetic example of Fig. 1.1, we arbitrarily assign the execution time of each node per homogeneous set of processor between 1 and 10. In this example, we map task Z to one of the processors from CPU 2

manually. The assumed profile of execution times for the tasks in the synthetic example is displayed in Table 5.1.



(a) MNIST



(b) K-means clustering

Figure 5.2: SDF/L benchmarks

The design space exploration results for these three benchmarks are displayed in Table 5.2. We compare the average time it takes to complete one iteration of the genetic algorithm. As expected, the proposed hierarchical scheduling is significantly faster than the flat baseline scheduling, from 7.9 times for K-means Clustering to 120 times for MNIST. Since the proposed approach is designed to take advantage of the loop structure, the speed-up gain depends on the ratio of the number of tasks inside the loop to the

Table 5.1: Execution time information of the synthetic example

Pool	Cores	Execution time (us)							
		S	D	A1	B1	C1	A2	B2	Z
CPU 1	4	5	7	4	6	7	3	4	3
CPU 2	2	3	5	2	5	5	3	4	1
GPU	1	-	4	1	3	2	1	2	1

Table 5.2: Design space exploration results

Application	Iteration Time (sec)		Resultant Latency (ms)	
	Flat	Hierarchical.	Flat	Hierarchical.
Synthetic	8.1	0.25	12.6	13.2
MNIST	263.5	2.2	10183.3	10182.9
K-means Clustering	50.0	6.3	87.5	87.5

number of tasks in the whole graph: The loop in K-means Clustering is a lot simpler than that in MNIST.

When comparing the latency results listed in Table 5.2, the flat scheduling found a little bit better schedule than the hierarchical scheduling technique for the synthetic benchmark. The lowest-level loop in Fig. 1.1 has four task instances per iteration: one for A2 and B2 each and two instances for Z. In the proposed technique, scheduling of the innermost graph schedules incurs empty time slots, called *slack*, due to unbalanced execution time between mapped processors and the tasks, due to this repeating the schedule as a whole in the upper-level scheduling degrades the utilization of processors. On the other hand, a flat scheduling approach schedules 25 instances of A2, B2, 50 instances of Z and all other tasks from other levels at once. Hence it could find a better schedule than the proposed schedule by avoiding the slack. The scheduling quality of the hierarchical scheduling depends on how well balanced is the loop schedule since the scheduling slack will begin piling up as loop iterations are added.

For the K-means Clustering benchmark, two scheduling techniques achieved the same latency results since there is only a single task inside the loop and there is no scheduling slack in the loop schedule. On the other hand, for MNIST benchmark, the

proposed hierarchical scheduling found a better schedule with a shorter latency than the flattened scheduling technique, which is counter-intuitive.

As the size of the flattened graph increases, flat scheduling becomes more difficult and may be too greedy to find a good solution for some graphs. If there is none or little slack in the loop schedule, considering it as a whole will make the upper-level schedule easier without affecting the performance. It explains why the hierarchical scheduling gives shorter latency than the flat scheduling for the MNIST benchmark.

In summary, the potential latency differences, which can go both ways, between two approaches are directly connected to the level of awareness of the scheduler and the scheduling slack of the loop schedule.

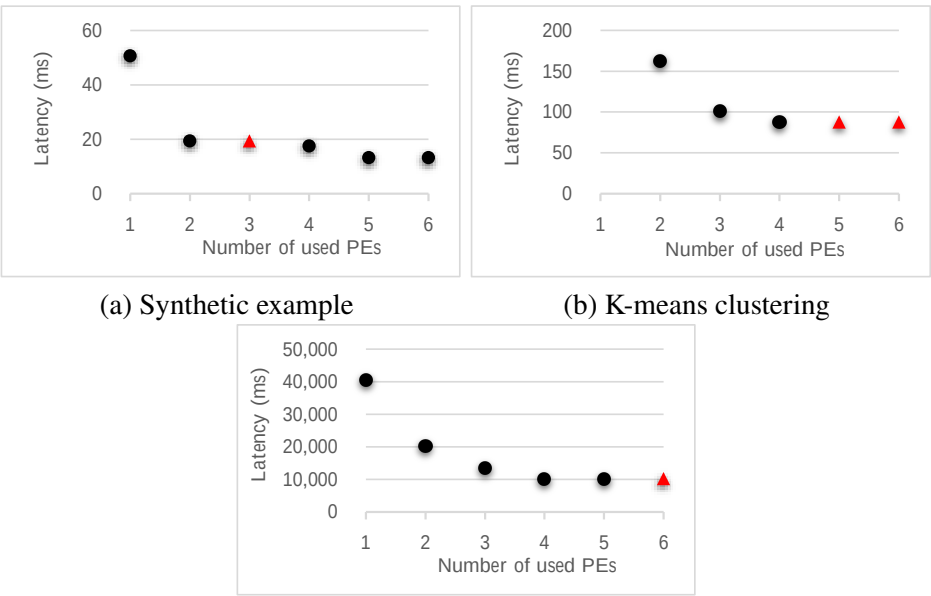
As mentioned earlier, we may want to minimize both the number processors used and the latency. By using a genetic algorithm, we can perform design space exploration (DSE) to obtain the Pareto-optimal solutions easily. The Pareto-optimal solutions after finishing DSE for the three examples are shown in Fig 5.3. The red triangles on the plots indicate solutions that are not Pareto-optimal.

Note that there is no optimal 3 processor mapping found for the synthetic example shown in Fig 5.3(a) as the best 3 processor mapping resulted in a schedule that had latency slightly worse than the 2 processor mapping. This comes specifically from the chosen profile and processing unit setup. As task Z is fixed to a processor in CPU 2 set, that processor should be included in any schedule we generate. This means if we want a 3 processor schedule, L1, L2 can be mapped to two processors from CPU 2 and GPU can be used for D. For any schedule using 3 processors, our loops are actually mapped to 2 processors which explains why we can not get a better schedule using 3 processors over using 2 processors. While the difference between 5 processor and 6 processor schedule latency is also very small, there is still a slight improvement with the added processor and both are considered Pareto-optimal.

The result shows that increasing the number of processors can improve the latency but

the benefit gain slows down as more processors are added and after some point might stop improving completely - this is the case with K-means example results for which are shown on Fig 5.3(b). The same is true for MNIST example which shows no improvement when adding the sixth processor, as shown on Fig 5.3(c). This is in part due to the restriction that loop mapping is limited to a set of homogeneous processors and the biggest set of homogeneous processors for this example contains 4 PEs.

For the K-means Clustering benchmark, there is no single processor Pareto-optimal mapping. It is because the number of generations, 100, is so small that no chromosome is made that maps all of the tasks to a single processor in our experiment: The probability of randomly generating a mapping that puts all tasks to same processor is low. Another setup where single processor mappings can not be tested would be one where multiple dedicated tasks are mapped to different processors. In this case the lowest number of processors in a valid mapping would correspond to the number of such dedicated processors.



(a) Synthetic example

(b) K-means clustering

(c) MNIST

Figure 5.3: Pareto-optimal solutions for three benchmarks.

5.2 Randomly Generated Graphs

Given the lack of real SDF/L examples currently implemented, we perform experiments with randomly generated graphs in order to give a more generalized overview on the performance of the hierarchical scheduling approach. For these experiments we assume a Jetson TX2-like architecture that has 3 pools of 1, 2 and 4 processors, respectively: The tasks can be mapped to 3 sets of homogeneous processors made up of 7 processors in total.

For each graph included in the experiment the whole GA based design space exploration is executed 10 times for both algorithms and the best results are recorded. This is to account for cases where we randomly start from an unfavorable initial generation for either of the algorithms and thus could potentially show one of the approaches more favorably than it actually is. Note that that the range of settings is limited by the effectiveness of flat scheduling algorithm as the SDF/L graphs have to be simple enough for both approaches to complete in a feasible time.

At the base the randomly generated graph has a number of fixed characteristics. A graph has anywhere between 5 to 25 tasks. A graph may have 1 to 3 D-type loops with a loop count in the range of 20 to 200. In case of multi-level loops (D-type loop inside D-type loop), the parent loop count is set to 10% of the inner loop count. The depth of the loop nest is limited to 2 to control the complexity of the graph in favor of the flattened scheduling approach. We assign tasks to D-type loops first. Then we make connections between tasks at each level. At the lowest level, we consider a task one by one by making a connection between a new task to a randomly chosen task to make a connected graph. In addition, we make more connections: for each ordered pair of tasks an additional connection is formed with 10% likelihood to avoid only generating completely linear SDF/L graphs. There is also a 5% chance that a feedback arc is added to the graph. At the upper level, a D-type loop is considered as a single task, and the same process of making connections among tasks is followed. After the task graph is constructed, we

Table 5.3: Improvement over the flat scheduling approach for randomly generated SDF/L graphs

	Latency improvement	Time improvement
WORST	-21.63%	107.21%
AVG	-2.46%	1913.94%
BEST	20.53%	19637.12%

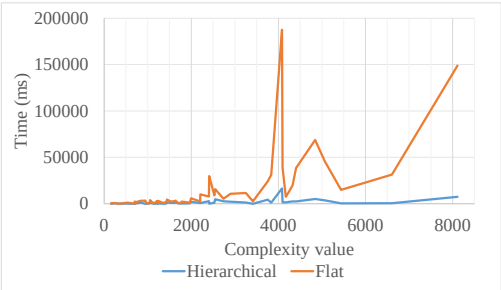


Figure 5.4: Comparison of average iteration time

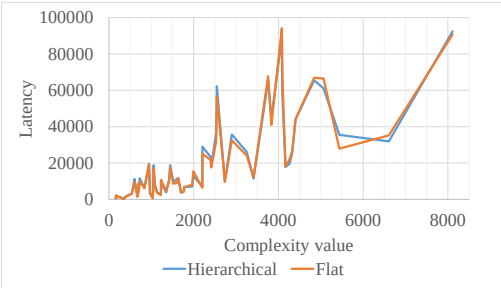


Figure 5.5: Comparison of latency

Figure 5.6: Randomly generated graphs schedule latency and iteration time comparisons

assign each task a repetition count between 1 to 5. In addition, there is a 5% chance that a task is fixed to a designated processor prior to the DSE algorithm being executed.

A total of 60 randomly generated SDF/L graphs were used for the experiments. Table 5.3 displays the summary of the comparison - the best, worst and average percentage of improvement when using the proposed hierarchical scheduling over the flat scheduling approach for the same randomly generated SDF/L graphs.

The number of total scheduled tasks in the final schedule can be used as a metric to assess the complexity of the graph. This value can also be calculated by Eq.(5.1) where t_i is the $i-th$ task out of n total tasks, p_i represents the parent (loop) task of t_i, and $LoopCount(p_i)$ is the loop count of the loop represented by task p_i.

$$\sum_{i=0}^{n-1} \begin{cases} 0, & \text{if } t_i \text{is a loop task} \\ Rep(t_i) * Rep(p_i) * LoopCount(p_i), & \text{if } t_i \text{is in loop } p_i \\ Rep(t_i), & \text{otherwise} \end{cases} \tag{5.1}$$

Figure 5.4 plots the average iteration times of both approaches for random graphs with a wide range of complexity defined above.

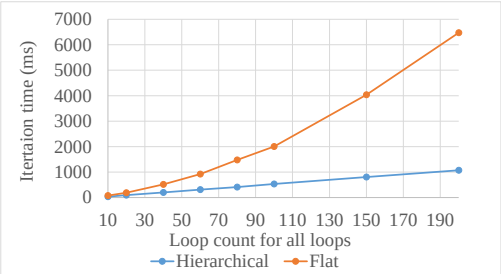
As we can see, the hierarchical scheduling has a significant advantage in terms of the iteration time over flat scheduling, regardless of the complexity level. Figure 5.5 shows the latency found for the same graphs. While the variance of latency difference is relatively small, the hierarchical scheduling can produce a better or worse latency result, similarly to the results with the benchmark experiments.

On average a slight loss in latency is expected, the difference in latency in our experiment set ranged from -21.63% to +20.53% with the average being -2.53%. In turn there is a significant improvement in speed of calculation ranging from 107.21% to 19637.12% of improvement. The average improvement over these 60 graphs included is 1913.94%.

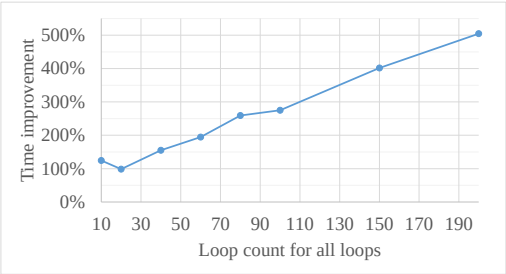
Since the speed-up of the hierarchical scheduling is dependent upon the complexity of the loops, not the complexity of the entire graph, the speed-up improvement over the flat scheduling is quite random in Figure 5.4. So we focus on the loop complexity that grows when the number of tasks in the loop, the repetition count of the loop task or the loop count grows.

Since the easiest setting to manipulate while minimally changing the base graph is the loop count, we generate a random graph and vary the loop count, ranging from 10 to 200.

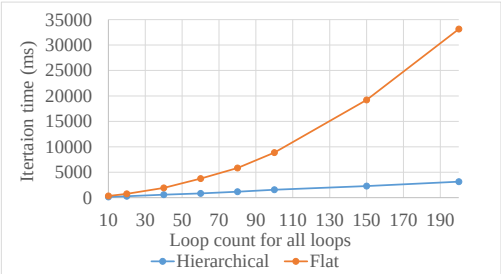
Experimental results for 2 such random graphs, graphs A and B, are displayed in Fig. 5.7 for the average iteration time, as the loop count increases, and the corresponding speed-up gain when comparing the two approaches. Note that the loop complexity of



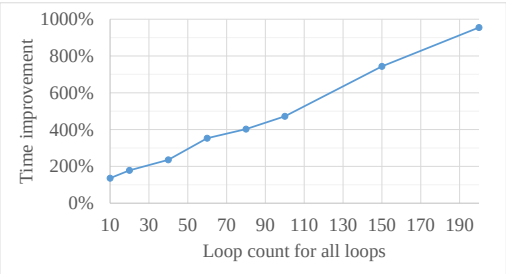
(a) Average iteration times for random graph A



(b) Comparative speed improvement for random graph A



(c) Average iteration times for random graph B



(d) Comparative speed improvement for random graph B

Figure 5.7: Iteration time improvement over the loop count increase for two random graphs

graph B is 1.8 times larger than graph A with the same loop count.

The results suggest, that as the loop complexity grows either through increasing the loop count (or the number of task instances inside the loop), the benefit margin of using the hierarchical scheduling technique over the baseline grows at a increasingly higher rate. From this we conjecture that with more complex graphs or higher loop counts that are not included in our experiment sets due to lacking capabilities of the baseline scheduling, the improvement of using the hierarchical scheduling will increase.

From the graphs we can see that the improvement percentage starts steadily increasing either right from the start or after increasing the loop count a few times. The point where the improvement ratio starts increasing is directly related with the number of unique patterns in the loop schedule and the horizontal extension potential of the schedule. As

stated in Section 4.3, when creating a hierarchical schedule for a loop, we schedule the loop iteration by iteration until finding a pattern and then extend the found pattern up to the loop count. If the pattern is not found before the loop count is met, which might occur when the loop count is small, we do not get any benefit of extension. As the loop count increases, the potential to find a pattern earlier grows and thus the hierarchical becomes more effective. The reason for the initial drop in Figure 5.7(b) for random graph A is caused by this: a repeating pattern is not found early until the loop count increases to 10. Once the loop count passes the unique iteration shapes count, the improvement over the flat scheduling gets increasingly better.

Combining both results from random graphs and concrete benchmark examples, we claim that using the hierarchical scheduling approach to compose the schedule is always noticeably faster than flat scheduling with a small potential variance in latency. This makes it feasible to use the GA-based approach for larger graphs where the baseline scheduling of a flattened graph is too slow to process a large number of potential mappings to find the favorable mapping in an affordable time.

Chapter 6

Conclusions

In the presented work, we introduced the problems and challenges related to SDF/L graph scheduling, when considering systems with heterogeneous processors. While the SDF/L model takes care of explicitly showing the data-parallelism, flattening the graph out for scheduling, fails to fully utilize the potential it provides. As seen, however, scheduling data-parallel loop tasks with high repetition and multiple mapping possibilities on complex systems while preserving, and exploiting, the hierarchical structure is not trivial.

In order to tackle these problems, we adopted a hierarchical scheduling methodology to map and schedule an SDF/L graph onto multiple heterogeneous processors. We present a number of small adjustments to the base scheduling methodology, a priority assignment scheme and a new hierarchical scheduling heuristic to cope with and take advantage of the hierarchical structure to more effectively utilize the potential that SDF/L model introduces.

The efficient mapping space for data-parallel tasks is defined to explore fast via a genetic algorithm. The effectiveness of the proposed method is verified with two real-life benchmarks, a synthetic example, and a number of randomly generated SDF/L graphs. While the real-life benchmarks used for the experiments might be on the simple side, the variety of random graphs, with different complexity levels used, serve to show the potential and scalability of the described approach. Based on the results, we claim that

scheduling SDF/L graphs hierarchically, following proposed methodology, the schedule generating process finishes significantly faster with the loss in latency being relatively small.

Bibliography

[1] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[2] Hyesun Hong, Hyunok Oh, and Soonhoi Ha. Hierarchical dataflow modeling of iterative applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17. Association for Computing Machinery, 2017.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[4] Avinash Malik and David Gregg. Executing synchronous data flow graphs on heterogeneous execution architectures using integer linear programming. *Trinity College Dublin, Tech. Rep*, 2012.

[5] Hoeseok Yang and Soonhoi Ha. Ilp based data parallel multi-task mapping/scheduling technique for mpsoc. In *2008 International SoC Design Conference*, volume 01, pages I–134–I–137, 2008.

[6] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374, 2006.

[7] Hoeseok Yang and Soonhoi Ha. Pipelined data parallel task mapping/scheduling technique for mpsoc. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 69–74, 2009.

[8] Hanwoong Jung, Hyunok Oh, and Soonhoi Ha. Multiprocessor scheduling of an sdf graph with library tasks considering the worst case contention delay. In *2016 14th ACM/IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–10. IEEE, 2016.

[9] Shin-haeng Kang, Duseok Kang, Hoeseok Yang, and Soonhoi Ha. Real-time co-scheduling of multiple dataflow graphs on multi-processor systems. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[10] Athena Abdi and Hamid R. Zarandi. A meta heuristic-based task scheduling and mapping method to optimize main design challenges of heterogeneous multiprocessor embedded systems. *Microelectronics Journal*, 87:1–11, 2019.

[11] Dowhan Jeong, Jangryul Kim, Mari-Liis Oldja, and Soonhoi Ha. Parallel scheduling of multiple sdf graphs onto heterogeneous processors. *IEEE Access*, 9:20493–20507, 2021.

[12] Bo Yuan, Xiaofen Lu, Ke Tang, and Xin Yao. Cooperative coevolution-based design space exploration for multi-mode dataflow mapping. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(3):1–25, 2021.

[13] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access*, 8:43980–43991, 2020.

[14] G.C. Sih and E.A. Lee. Declustering: a new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):625–637, 1993.

[15] Sardar M. Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 357–368, New York, NY, USA, 2011. Association for Computing Machinery.

[16] G.N. Srinivasa Prasanna, A. Agarwal, and B.R. Musicus. Hierarchical compilation of macro dataflow graphs for multiprocessors with local memory. *IEEE Transactions on Parallel and Distributed Systems*, 5(7):720–736, 1994.

[17] S. Ha and E.A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7):768–778, 1997.

[18] Adnan Bouakaz. *Real-time scheduling of dataflow graphs*. PhD thesis, Université Rennes 1, 2013.

[19] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, ed-

itors, *Euro-Par'96 Parallel Processing*, pages 573–577, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[20] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[21] Yuhei Suzuki, Takuya Azumi, Nobuhiko, Nishio, and Shinpei Kato. Hlbs: Heterogeneous laxity-based scheduling algorithm for dag-based real-time computing. In *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 83–88, 2016.

[22] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[23] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, page 401–410, New York, NY, USA, 2006. Association for Computing Machinery.

[24] Suleyman Tosun, Nazanin Mansouri, Mahmut Kandemir, and Ozcan Ozturk. An ilp formulation for task scheduling on heterogeneous chip multiprocessors. In Albert Levi, Erkay Savaş, Hüsnü Yenigün, Selim Balcısoy, and Yücel Saygın, editors, *Computer and Information Sciences – ISCIS 2006*, pages 267–276, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[25] Sanjit Kumar Roy, Rajesh Devaraj, Arnab Sarkar, Kankana Maji, and Sayani Sinha. Contention-aware optimal scheduling of real-time precedence-constrained task graphs on heterogeneous distributed systems. *Journal of Systems Architecture*, 105:101706, 2020.

[26] Pranav Tendulkar, Peter Poplavko, Jules Maselbas, and Oded Maler. Pipelined scheduling of acyclic sdf graphs using smt solvers. In *The International Workshop on Investizating Dataflow in Embedded computing Architecture*, 2015.

[27] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices*, 41(11):151–162, 2006.

[28] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, page 179–196, Berlin, Heidelberg, 2002. Springer-Verlag.

[29] Svetlana Minakova, Erqian Tang, and Todor Stefanov. Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In *International Conference on Embedded Computer Systems*, pages 18–35. Springer, 2020.

[30] Panayiotis Alefragis, Christos Gogos, Christos Valouxis, George Goulas, Nikolaos Voros, and Efthymios Housos. Assigning and scheduling hierarchical task graphs to heterogenous resources. *Proceedings of 10th Practice and Theory of Automated Timetabling (PATAT 2014)*, 08 2014.

[31] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4j: a modular framework for meta-heuristic optimization. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1723–1730, 2011.

[32] Hope Of Parallel Embedded Software development (HOPES). *SDF/L examples*, 2020.

요 약

이종 멀티 코어 프로세서에서 SDF/L 그래프 스케줄링 기법

Mari-Liis Oldja
공과대학
컴퓨터공학부
서울대학교 대학원

데이터플로우 모델은 애플리케이션의 태스크를 병렬 처리할 때 좋은 모델로 알려져 있지만 데이터를 병렬로 처리하는 데에 활용하기는 어렵다. 데이터 수준 병렬 처리는 루프 구조를 통해 표현될 수 있으나 기존 데이터플로우 모델에서 명시적으로 루프 구조는 명세하는 방법이 없었다. 이러한 단점을 극복하기 위해 계층적 구조를 활용하여 루프 구조를 명시적으로 명세할 수 있는 SDF/L 모델이 제안되었다. 그러나 이기종 프로세서에 대한 SDF/L 그래프의 스케줄링은 이전까지 고려되지 않은 것으로 파악된다.

본 논문에서는 SDF/L 모델로 표현되는 애플리케이션을 이기종 프로세서에 대하여 스케줄링하는 기법을 소개한다. 제안된 방법에서는 먼저 진화적 메타 휴리스틱을 사용하여 태스크 매핑을 탐색한다. 이후 하위 수준에서 병렬 루프 스케줄을 만든 다음 상위 수준에서 스케줄 구성할 때 재사용하는 상향식의 계층적 태스크 스케줄링을 수행한다. 제안하는 스케줄링 기법의 효율성을 검증하기 위해 벤치마크 예제와 무작위로 생성된 SDF/L 그래프에 기법을 적용하였다.

주요어 : 매핑 및 스케줄링, 계층적 스케줄링, 데이터 병렬 스케줄링, 데이터플로우 아키텍처, 설계 공간 탐색

학번 : 2019-24125

본 논문작성자는 한국정부초청장학금(Global Korea Scholarship)을 지원받은 장학생임