Application Mapping on Multiprocessor Hardware

Platforms using

Genetic Algorithms

By

DONGZHE SU

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Master of Philosophy
in Computer Science and Engineering

June 2009, Hong Kong
Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Su Dongzhe
DONGZHE SU
Application Mapping on Multiprocessor Hardware

Platforms using

Genetic Algorithm

By

DONGZHE SU

This is to certify that I have examined the above M.Phil thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Gu Zonghua
Dr. ZONGHUA GU. THESIS SUPERVISOR

Prof. Mounir HAMDI, HEAD OF DEPARTMENT

Department of Computer Science and Technology

June 2009
ACKNOWLEDGEMENT

First and foremost, I would like to give my deepest thanks to my supervisor Prof. Zonghua Gu, for all his guidance, support, comments and essential help while I’m purchasing my MPhil degree. I’m always impressed by his passion for research and his consideration towards students.

I would like to thank Weichen Liu, for his important comments and inspiration on my research work.

I also like to thank the committee members of my thesis, Prof. Charles Zhang and Prof. Lin Gu, who took the time to read my thesis and offer important comments.
# TABLE OF CONTENTS

Title Page i
Authorization Page ii
Signature Page iii
Acknowledgments iv
Table of Contents v
List of Figures vii
List of Tables viii
Abstract ix

## Chapter 1 Introduction

1.1 SDF Graph 1
1.2 Homogeneous SDF 3
1.3 Our work 4

## Chapter 2 Related work

2.1 Overlapped Schedule 5
2.2 Scheduling strategy 5
2.3 Resource constraints 7
2.4 Exact Solution Technique 8

## Chapter 3 Mapping and Scheduling

3.1 Application architecture 9
3.2 Actor mapping and scheduling 10
   3.2.1 Buffer constraint and Network delay 10
   3.2.2 Scheduling 12

## Chapter 4 Optimization Approaches

4.1 Encoding 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1 Encoding of mapping</td>
<td>16</td>
</tr>
<tr>
<td>4.1.2 Encoding of scheduling</td>
<td>17</td>
</tr>
<tr>
<td>4.1.3 Encoding of an individual</td>
<td>17</td>
</tr>
<tr>
<td>4.1.4 Genes of an individual</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Optimization Procedure with GA</td>
<td>18</td>
</tr>
<tr>
<td>4.2.1 Computing the MCM for each individual</td>
<td>19</td>
</tr>
<tr>
<td>4.2.2 Architecture aware HSDF graph</td>
<td>20</td>
</tr>
<tr>
<td>4.2.3 Important Parameters in GA</td>
<td>21</td>
</tr>
<tr>
<td>4.2.4 Recombination</td>
<td>21</td>
</tr>
<tr>
<td>4.2.5 Mutation of Genes</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Enhancement to the search Procedure</td>
<td>24</td>
</tr>
<tr>
<td>Chapter 5 Experiment and Comparisons</td>
<td>26</td>
</tr>
<tr>
<td>5.1 Experiment setup</td>
<td>26</td>
</tr>
<tr>
<td>5.2 Experiment results</td>
<td>26</td>
</tr>
<tr>
<td>5.3 Discussions</td>
<td>29</td>
</tr>
<tr>
<td>Chapter 6 Conclusions</td>
<td>31</td>
</tr>
<tr>
<td>References</td>
<td>32</td>
</tr>
</tbody>
</table>
List of figures

Figure 1 A SDF graph example........................................................................................................ 1
Figure 2 A multiprocessor platform consist of 4 processors ......................................................... 9
Figure 3 Effect of buffer constraint.............................................................................................. 10
Figure 4 HSDF without auto-concurrency..................................................................................... 11
Figure 5 HSDF with network latency ............................................................................................ 12
Figure 6 Platform independent HSDF ......................................................................................... 13
Figure 7 HSDF graph with static ordering BCD on P2 ................................................................. 13
Figure 8 HSDF graph with static ordering BDC on P2 ................................................................. 14
Figure 9 Example of mapping........................................................................................................ 16
Figure 10 Basic workflow of GA.................................................................................................... 19
Figure 11 Example to illustrate the need to check actor ordering constraints before GA..................... 25
Figure 12 Comparison of the MCM values obtained with GA...................................................... 27
Figure 13 Comparison of Algorithm running times......................................................................... 28
List of Tables

Table 1 Change of the tokens on edges during execution of the schedule AABAB for the SDF graph in Figure 1 .............................................................. 2

Table 2 Experiment results .................................................................................. 27
Application Mapping on Multiprocessor Hardware

Platforms using

Genetic Algorithm

by

DONGZHE SU

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

Abstract

Synchronous Dataflow (SDF) is a widely-used model-of-computation for signal processing and multimedia applications. As Moore’s law comes to an end, multiprocessor systems are becoming ubiquitous in today’s embedded systems design. In this thesis, I address the problem of mapping a Homogeneous Synchronous Dataflow (HSDF) graph onto a multiprocessor platform with the objective of maximizing system throughput. The hardware platform consists of multiple processors connected with a communication substrate with guaranteed inter-processor communication latency, e.g. a hard real-time Network-on-Chip. Since the problem is a NP-hard combinatorial optimization problem, it is generally infeasible to use exhaustive search to obtain optimal solutions for realistic-sized
applications. In this thesis, I adopt Genetic Algorithms, to search the design space of all possible actor-to-processor mappings and actor orderings on each processor to find the optimal solution, and compare the performance and scalability of GA with the exact solution technique based on SAT solving.
CHAPER 1

INTRODUCTION

1.1 SDFG

In the dataflow paradigm, a program is represented as a directed graph, where nodes, called actors, represent computational modules, and directed edges represent communication channels between the modules. There are a variety of dataflow models, including Synchronous Dataflow (SDF) (also called Statically Schedulable Dataflow (SSDF)), Cyclo-Static Dataflow (CSDF), Multi-Dimensional SDF (MDSD), Boolean Dataflow (BDF), and others. In this thesis, we focus on Synchronous Dataflow (SDF), which is a special type of dataflow model where each actor invocation consumes and produces a constant number of data tokens. It is widely used in a broad class of signal processing and digital communication applications, including modems, multi-rate filter banks, and satellite receiver systems. A SDF graph can be statically scheduled offline because of its static nature.

Figure 1 A SDF graph example
As an example, Fig. 1 shows a simple SDF graph. Each firing of actor A consumes 2 tokens on edge $e_{BA}$, and produces 2 tokens on edge $e_{AB}$; each firing of actor B consumes 3 tokens on edge $e_{AB}$, and produces 3 tokens on edge $e_{BA}$. We can view actor A as the producer, and actor B as the consumer, connected by edge with buffer size 4. Number of tokens on edge denotes the number of empty spaces. Or we can view actor B as the producer, and actor A as consumer, connected by edge with buffer size 4. The two views are symmetric and equivalent from the perspective of SDF theory. A feasible static schedule is AABAB, i.e. starting from the initial state in Fig 1, if the actors are invoked in the sequence AABAB, then the SDF graph goes back to the initial state. Therefore, we can execute this sequence of actor firings repeatedly without any deadlock or buffer overflow conditions. Table 1 shows how the number of tokens on the 2 edges in the SDF graph evolves during execution of the schedule AABAB.

Formally, a SDF graph is defined as $G = (V, E)$. It is a directed graph with a set of nodes $V = \{v_i\} \ 1 \leq i \leq n$ which representing the actors, combined with a set of directed edges $E = \{e_{ij}\} \ i, j \in [1, \ldots, n]$ representing the communication channels from source actor $v_i$ to sink actor $v_j$. When an actor $v_i$ fires, the

<table>
<thead>
<tr>
<th></th>
<th>Init</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_{BA}$</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$e_{AB}$</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1 Change of the tokens on edges during execution of the schedule AABAB for the SDF graph in Figure 1
number of tokens it consumes (produces) on each input (output) edge $e_{ij}$ is fixed and known at compile time, denoted as $cons(e_{ij}) (prod(e_{ij}))$. Each channel $e_{ij}$ has a known buffer size $buff_{ij}$, representing the maximum number of tokens it can store at one time. Each channel $e_{ij}$ may contain a number of initial tokens, also called delays, denoted as $d_{ij}$. A simple cycle is a cycle in the SDF graph where no node appears more than once.

In this thesis, we consider a homogeneous computing environment, where all processors are identical in terms of processing speed, hence each actor $v_j$ has the same execution time on different processors, and $v_j$ has a known Worst-Case Execution Time (WCET) $t_j$ that is independent of the processor that it is mapped to.

### 1.2 Homogeneous SDFG

A HSDF graph is a special SDF graph that

$$cons(e_{ij}) = prod(e_{ij}) = 1$$

for all communication channels. Every SDF graph can be converted to its equivalent HSDF graph, although the size of the HSDF graph may be much larger than the original SDF graph. The cycle mean of a cycle $c$ in a HSDF graph is defined as

$$\mu_c = \sum_{i \in N(c)} t_i / \sum_{e \in E(c)} d_e,$$

where $N(c)$ is the set of all nodes traversed by cycle $c$, and $E(c)$ is the set of all edges traversed by cycle $c$. The Maximum Cycle Mean (MCM) of a HSDF graph $G$ is defined as:

$$\mu(G) = \max_{c \in C(G)} \left( \frac{\sum_{i \in N(c)} t_i}{\sum_{e \in E(c)} d_e} \right)$$  

(1)

Where $C(G)$ is the set of simple cycles in graph $G$; $N(c)$ is the set of all nodes traversed by cycle $c$, and $E(c)$ is the set of all edges traversed by cycle $c$. A cycle is
called the critical cycle if it has the largest MCM among all simple cycles in graph G. It is well known that the maximum throughput of a HSDF graph is the inverse of its MCM; hence maximizing throughput of a HSDF graph is equivalent to minimizing its MCM. There may be multiple critical cycles which have the same MCM in a graph G. There are efficient polynomial-time algorithms for computing the MCM of a HSDF graph [2]. A deadlock cycle is a simple cycle with no initial tokens, which should be prevented during actor mapping and ordering.

1.3 Our Work

Definition 1 Problem Formulation:
Consider a HSDF graph with fixed buffer size on each edge, and a homogenous multi-processor platform with fixed inter-processor communication delay and unlimited bandwidth. Actor scheduling algorithm on each processor is non-preemptive static-order scheduling. Find an actor-to-processor mapping and static ordering among actors on the same processor with the maximum possible system throughput.

The overall optimization objective is to maximize system throughput. There are three steps to solving the optimization problem:

1. Map each actor to a single processor
2. Find a static ordering for actors mapped onto the same processor
3. Perform deadlock detection, and calculate the MCM for a given deadlock-free mapping and ordering.

We adopt Genetic Algorithms to explore the design space, which consists of all possible actor-to-processor mappings and actor orderings on each processor.
CHAPTER 2
Related Work

2.1 Overlapped Schedule

There are two types of multiprocessor schedules: non-overlapped schedules and overlapped (pipelined) schedules. For non-overlapped schedules [3, 4], different iterations of the SDF graph execution cannot overlap in time. Task graphs are used to model the application, and the objective is to minimize the overall schedule length (makespan). Task graphs can be viewed as a special case of HSDF graphs without any initial delay tokens, hence they should not contain any cycles, since any cycle will lead to a deadlock. In contrast, cycles are allowed in HSDF as long as there are non-zero initial tokens in the cycle, which means that actor firing in HSDF does not have to follow the edge precedence order. We focus on overlapped schedules in this thesis, which can achieve higher throughput than non-overlapped schedules by exploiting inter-iteration concurrency.

2.2 Scheduling Strategy

It is known [5] that if the number of processors is unlimited, the optimal overlapped schedule of a SDF graph with maximum throughput can be computed in polynomial time. However, when the number of available processors is less than the application’s maximum degree of parallelism (which is the maximum number of actors that can be enabled concurrently), we need to find a scheduling strategy to order the firing of
those actors that share the same processors, hence the problem of mapping and scheduling of an SDF Graph on a multiprocessor platform becomes NP-hard.

There are several scheduling algorithms [6]: Time-Division Multiple Access (TDMA), static-order, round-robin, fixed-priority and dynamic-priority scheduling. TDMA scheduling is typically used for scheduling between different applications, and static-order scheduling is typically used for scheduling actors within the same application, where a cyclic actor firing sequence is defined offline, and the scheduler simply fires each actor following the predefined sequence.

In [6], mapping of multiple time-constrained applications is studied. Using SDFG to model each one of the applications, the optimization objective is to minimize the resource usage (processor space, memory and communication bandwidth) while meeting a user-specified throughput constraint. TDMA is used to guarantee the execution time of the actors (possibly from different applications), when binding them to the same processor. TDMA uses a periodically rotating time wheel, and an application reserves a time slice on the wheel to fire its actors. This provides performance isolation among applications. And then, static order scheduling is used to schedule the actors of the same application. Under the given throughput constraint, the goal is to reduce the resource usage of each application (number of time slices it occupies on the time wheel) in order to maximize the number of applications that can be run on the same shared hardware platform.

In [6], the authors used *list-scheduling* to construct the static ordering. When an actor becomes enabled in the execution of an SDF graph, it does not start its firing immediately. Instead the actor is added to the ready list of the tile (processor) that it is bounded to. When no actor is firing on the tile, the first actor is removed from the list and its firing starts. At this time, the actor is added to the schedule of the tile.
After finding a recurrent state, a finite length schedule is generated for the tile

In [7], a problem similar to [6] is addressed. TDMA and static scheduling are combined to schedule multiple real-time applications to a homogeneous multiprocessor platform. The optimization objective is to minimize the resource usage while meeting the throughput constraint of each application. The author uses exhaustive search on top of a list scheduler for static order scheduling; and linear programming to assign TDMA time wheel sizes.

2.3 Resource constraints

[5] presented an efficient state space exploration technique for calculating the Pareto space of throughput and storage trade-offs, which can be used to determine the minimal buffer space needed to execute a graph under a given throughput constraint, with the assumption that the number of processors is unlimited. Under this assumption, the optimal schedule of an SDF graph with maximum throughput can be computed in polynomial time without converting to HSDF.

As there are no resource constraints, all enabled actors in SDF graph can and will fire if their dependency constraints are satisfied. The firing rule is a polynomial-time deterministic algorithm that is guaranteed to terminate or result in a deadlock. Nevertheless, the unlimited resource requirement may not hold in many applications. When the number of processors is less than the maximum concurrently enabled actors, we have to schedule the fire sequence of the actors, and the problem becomes NP-hard, as there are exponentially number of scheduling choices to try in order to find the optimal solution.
2.4 Exact Solution Technique

For some cost-sensitive real-time applications, it may be important to search all possible scheduling and mapping choices to find an optimal solution. [1] presented an exact solution technique based on SAT solver for mapping a HSDF graph onto a multiprocessor platform with the objective of maximizing system throughput. Based on branch-and-bound and SAT-solving techniques, two optimization approaches are presented: Logic-Based Benders Decomposition (LBBD) approach, and the integrated approach.

The integrated approach integrate branch-and-bound search into the SAT engine to achieve effective search tree pruning and better scalability. Although it returns exact optimal solutions to the mapping and scheduling problem and scales better than the LBBD approach, the integrated approach still has limited scalability: when the number of processor and the size of the HSDF graph are relatively large, the running time of the algorithm can be extremely long.
3.1 Application Architecture

We consider a homogenous multiprocessor platform consisting of multiple identical processors connected with a communication substrate with guaranteed latency.

Definition 2 Architecture Graph

Architecture Graph $\text{ACG} (P, L)$ is an undirected graph consists of $|P|$ identical processors. $P = \{ p_1, p_2, p_3, \ldots, p_n \}$, where $|P| = n$ is the number of processors.

$L = \{ l_{ij} | i, j \in \{1, \ldots, n\} \}$, $l_{ij}$ represents the link between $p_i$ and $p_j$.

![Figure 2 A multiprocessor platform consist of 4 processors](image)

Figure 2 is an example multiprocessor platform consisting of 4 identical processors.

So formally its $\text{ACG}(P, L)$, where $P = \{ p_0, p_1, p_2, p_3 \}$, $L = \{ l_{01}, l_{02}, l_{13}, l_{23} \}$. 
3.2 Actor Mapping and Scheduling

For a given hardware platform and a given application HSDF graph, different actor-to-processor mappings and static-order scheduling on each processor will lead to different system throughput. To find the mapping and scheduling with maximum throughput, we generate a new platform-specific HSDF graph from the original platform-independent HSDF graph for each possible mapping and scheduling. In order to identify this problem formally, we add additional actors and edges to the HSDF graph to model platform elements such as buffer size, network delay and CPU. We borrow from the Model-Driven Architecture (MDA) terminology to refer to the application HSDF as the Platform-Independent Model (PIM), and the HSDF enhanced with platform elements as the Platform-Specific Model (PSM), which contains additional actors and edges to model platform elements, including network actors for modeling network delays. In the following figures, network actors are drawn as shaded circles in order to distinguish them from application actors.

3.2.1 Buffer constraint and Network delay

Figure 3 Effect of buffer constraint

Figure 3(a) shows a PIM HSDF graph without any platform constraint. WCET(A) = 10, WCET(B) = 6. Its MCM is undefined since it does not contain any cycles, hence
its maximum throughput is unlimited in theory, since it can grow without an upper bound with increasing number of available processors. If we add a buffer-size constraint to Figure 3(a) by adding one token on the input channel of actor A, the HSDF goes to 3(b). Now there is a cycle with 1 token on it, and its MCM is \((10+6)/1 = 16\). The buffer-size constraint means that actors A and B must fire serially, hence adding more processors will not help increase system throughput.

![Figure 3(a) and 3(b)](image)

**Figure 4**  HSDF without auto-concurrency

In Figure 4(a), we prevent *auto-concurrency* by adding a self channel on each actor with one token on it. This means that an actor must finish one firing before starting a new firing. In this case, the MCM of Figure 4(a) will be 10 \(\{\text{WCET (A)}/1 = 10\} > \{\text{WCET (B)}/1 = 6\}\). In Figure 4(b), we add an additional buffer-size constraint of 2 tokens on edge AB. The MCM is still 10 because the newly-added cycle has a CM of \((10+6)/2 = 8\), which is smaller than 10. Hence the newly-added cycle is not a critical cycle, and the additional buffer-size constraint does not affect the maximum system throughput.
Figure 5 (b) adds network actor N between actors A and B to denote network delay between the two processors that actors A and B are mapped to. The self-edge on actor N means that only one token can be transmitted at any given time, i.e., concurrent token transmission is not permitted. If WCET (N) = 4, transmission of one token between A and B consumes 4 time units, and transmission of 2 tokens consumes 8 time units. The MCM of the new HSDF graph is now \((10+4)/1 = 14\).

### 3.2.2 Scheduling

There are several possible actor scheduling algorithms for a given SDF graph.

- Fully-static: in compile time, the exact firing time of each actor is determined.
- Self-timed: Each actor fires immediately when its firing precedence is satisfied.
- Static-order: The order of actor firing on the same processor is statically-determined at compile time.

Since each HSDF actor fires only once at each iteration, the scheduling problem is equivalent to finding a static-ordering of actors mapped to the same processor, and we should try all possible orderings in order to find the one with the smallest MCM. We use an example to explain the effect of static ordering on the MCM.
Figure 6 Platform independent HSDF

Figure 6 shows a platform-independent HSDF graph. Suppose there are two processors: P1 and P2. Actor A is mapped on P1, and actors B, C and D are mapped on P2. There are three possible static orderings: B>C>D, B>D>C, C>B>D. (We omit the network actor and only focus on actor scheduling on each processor). Suppose WCET of all actors are the same and equal to 5, then the PIM HSDF has MCM 15, with cycle ABD as the critical cycle.

Figure 7 HSDF graph with static ordering BCD on P2
We first consider the static ordering BCD on P2. In Figure 7, two new edges are added to the HSDF to construct the PSM: B to C, C to D. A new cycle ABCD is generated, and the MCM is now 20.

We then consider the static ordering BDC on P2. In Figure 8, the critical cycle is ABD and the MCM is 15.

This example shows that different static orderings can lead to different MCM values. If the self-timed scheduling policy is used, we will not be able to control the runtime ordering of actors B, C, and D, hence we have to assume the worst case and set the MCM to be the largest MCM among all the possible static orderings, i.e., 20. But using static-order scheduling, we can choose the static ordering BDC on P2 to obtain the MCM of 15.

Note that when we add edges to the HSDF graph to enforce a static ordering on each processor, we also need to add some back edges, each having 1 token on it, in order to make sure that the order stays static between different iterations of the HSDF execution. For example, in Figure 7, a back-edge DB is added; in Figure 8, a back-edge CB is added.
Since finding the mapping and ordering with the minimum MCM is a NP-hard combinatorial optimization problem, it is infeasible to obtain optimal solutions for realistic large-size applications with any exact solution technique. With a small number of processors and/or tasks, exhaustive search may be feasible by listing all the possible mapping and scheduling strategies, but the search space grows exponentially with problem size. In this thesis, I apply genetic Algorithms (GA) to solve this problem.

GA is a stochastic optimization algorithm based on the principles of biological evolution for solving multi-objective optimization problems. In this thesis, we focus on optimizing a single objective, the system throughput, but we can easily handle additional optimization objectives, e.g., buffer size. Next, we introduce the problem encoding.

### 4.1 Encoding

Consider an application HSDF: TG (V, E). It is a directed graph with a set of nodes V = {v_i | 1 ≤ i ≤ n} which representing the actors, combined with a set of directed edges E = {e_{ij} | i, j ∈ [1, … , n]} representing the communication channels from source actor v_i to sink actor v_j. |V| is the number of actors in TG. Each channel e_{ij} has a known buffer size buff{ij}, representing the maximum number of tokens it can
store at any given time. Each channel \( e_{ij} \) may contain a number of initial tokens, also called *delays*, denoted as \( d_{ij} \).

The Architecture Graph ACG (\( P, L \)) is an undirected graph consists of \(|P|\) identical processors. \( P = \{ p_1, p_2, p_3, \ldots, p_n \} \), where \(|P|\) is the number of processors.

\( L = \{ l_{ij} | i, j \in \{1\ldots n\} \} \), \( l_{ij} \) represents the link between \( p_i \) and \( p_j \).

### 4.1.1 Encoding of Mapping

We use \( |V| \left\lceil \log |P| \right\rceil \) bits to encode the mapping of actors to processors. \(|V|\) is the number of actors. Each actor can be mapped to any one of the \(|P|\) processors, so \( \left\lceil \log |P| \right\rceil \) bits are needed to store the mapping information for each of the actors.

In total, \( |V| \left\lceil \log |P| \right\rceil \) bits are needed to store all the mapping choice of all the actors.

![Figure 9 Example of mapping](image.png)

In the mapping example showed in Fig 9, where the platform consists of 4 processors (\(|P|=4\)), and there are 6 tasks in the application graph. We use 00, 01, 10, 11 to represent each of the 4 processors \( p_0, p_1, p_2, p_3 \). For example, the specific mapping
in Figure 6b. can be encoded as the string 00 01 10 01 11 11.

4.1.2 Encoding of Scheduling

We use $|V|*(|V|-1)$ bits to encode the scheduling matrix of the actors. For each pair $V_i, V_j, 1 < i < j < n$. $a_{ij}$ represent the ordering between $V_i$ and $V_j$.

- $a_{ij}=00$ means that $V_i$ precedes $V_j$.
- $a_{ij}=01$ means that $V_j$ precedes $V_i$.
- $a_{ij}=10$ means that $V_i$ and $V_j$ are mapped on two different processors, hence do not have any precedence relationship between them.
- $a_{ij}=11$ means that the ordering between $V_i$ and $V_j$ is fixed and cannot be changed.

For example, when $|V|=3$, and $|P|=1$. We are using $|V|*(|V|-1)=3*2=6$ bits to store the scheduling information. If the static ordering of the three actors is: $V_1$ precedes $V_3$, $V_3$ precedes $V_2$, this can be encoded as:

$$
\begin{pmatrix}
  a_{12} & a_{13} \\
  a_{23}
\end{pmatrix}
= \begin{pmatrix}
  00 & 00 \\
  01
\end{pmatrix}
$$

4.1.3 Encoding of an Individual

An individual that encodes a given mapping and scheduling consists of one part of genes with $|V|*\lceil\log|P|\rceil$ bits to encode the mapping, and another part with $|V|*(|V|-1)$ bits to encode the scheduling. Then in this way, an individual contains all the information about a single mapping and scheduling strategy.
4.1.4 Genes of an Individual

In a single individual, for the mapping part, every single actor has its own \( \lceil \log |P| \rceil \) bits gene to represent their position of mapping. For the scheduling part, every single pair of vectors uses a 2-bit gene to encode their ordering.

4.2 Optimization Procedure with GA

As shown in Fig 10, starting with the HSDF application graph TG \((V, E)\) and architecture graph ACG, an initial population of individuals are generated (for example, the population size can be 200 individuals at a time for GA).

For each individual, a Platform-Specific Model (PSM), also called an architecture-aware HSDF graph, is generated. If there is no deadlock, its MCM can be calculated and stored as the optimization objective for the individual. We then use GA to search the design space with the objective of minimizing the MCM. Each time a new individual is generated; it will generate a new PSM and compute its corresponding MCM value.
4.2.1 Computing the MCM for each individual

Computing the MCM is a subroutine invoked in the inner loop of the optimization procedure. For each newly-generated individual, we compute its MCM as its optimization objective.

Algorithm1 MCM calculation

```c
double eval_mcm(individual *ind) /* for each individual ind, calculate mcm*/
{
    newG = GenerateNewGraph (ind,originalG); /*generate PSM HSDF :newG*/
}
```
If deadlockfree(newG) then /*check for deadlock*/
    mcm = maximumCycleMean (newG);
    /*calculate the mcm of newG*/
    return (mcm);
else output ’deadlocked’
    return 3000;
}

The procedure maximumCycleMean() is the implementation of a known MCM-calculation algorithm. We set the maximum value of MCM to be 3000, so we return 3000 if the architecture-aware HSDF contains a deadlock, so that this individual is eliminated from consideration as a possible candidate solution.

### 4.2.2 Architecture aware HSDF graph

Algorithm2 Generate PSM HSDFG

GenerateNewGraph (map,comm,ord, HSDFgraph *g)
{
    addedge(map,ord,g); /* add edges to encode the ordering of actors running on the same processor*/
    addnode(map,comm,ord,g);
    /* add a communication actor between two actors with an edge between them but mapped to two different processors*/
    addnewedge(g); /* add edges to connect the communication actors to application actors*/
addselfedge(g);     /*add self edge on new nodes to disable auto-concurrency*/
deletechannel(g);    /*delete old channels*/
return(g);
}

Algorithm 2 shows the algorithm when generating an PSM HSDFG using mapping and scheduling information from an individual. The output of the algorithm will be used to calculate MCM of the individual. For details of the algorithm used in GenerateNewGraph(), please refer to Chapter 3.

4.2.3 Important Parameters in GA

GA use recombination of parents and chance of gene mutation to generate new individuals. Following are several important parameters used in GA.

Size of the initial population (alpha): we set alpha to 100. This parameter affects the maximum number of parents and children in GA and its running time.

Number of objectives (1): We set it to 1 since there is only one optimization objective (the MCM).

Maxgen (maximum number of generations in GA): This parameter can be set based on the number of actors. It is set to 20 in our experiments. It should be set to a larger value for larger application HSDF graphs.

4.2.4 Recombination

Recombination of individuals and mutation of gene are two basic technique used in GA.
Algorithm 3 One point-crossover:

```c
int one_point_crossover(individual *ind1, individual *ind2)
/* Performs "one point crossover". Takes pointers to two individuals as arguments
and replaces them by the children (i.e. the children are stored in ind1 and ind2). */
{
    int position, i;
    int *map_string_ind2;
    map_string_ind2 = (int *) malloc(sizeof(int) * ind2->num_T);
    for(i = 0; i < ind2->num_T; i++)
        map_string_ind2[i] = ind2->map_string[i];
    position = irand(ind2->num_T);  /*randomly pick a crossover point*/
    for(i = 0; i < position; i++) {
        ind2->map_string[i] = ind1->map_string[i];
        ind1->map_string[i] = map_string_ind2[i];
    }
    free(map_string_ind2);
    re_sch(ind1);   /*change the actor ordering*/
    re_sch(ind2);
    /* evaluate new inds */
    ind1->mcm = eval_mcm(ind1);
    ind2->mcm = eval_mcm(ind2);
    return(0);
}
```

In Algorithm 3, two new children individuals are generated from recombination of two parents’ genes. The crossover point is chosen on the level of mapping. This is
because the scheduling of actors on processors is generated depending on the mapping strategy. With mapping strategy changes, the scheduling of actors should be generated accordingly.

The ‘re_sch()’ function is used to protect a viable gene from changes through the recombination. For example, if actor A and B are mapped to the same processor and their ordering should not be changed during recombination. This is a method for protecting a good gene from being broken during the search process.

### 4.2.5 Mutation of Genes

Algorithm 4 shows one type of mutation: **one point mutation**. This type of mutation only changes 1 position of gene at a time. The impact of this type of mutation during the evolution process will be limited.

Another strategy of mutation is called **independent bit mutation**, as shown in
Algorithm 5. With the parameter ‘pro’, every single gene will be judged independently for conduct of mutation.

We can control the position of mutation in GA with the setting of ‘pro’. For example, if pro is set to 1. Then every single gene will be mutated in a generation. Mutation will take the lead in the evolutionary strategy.

Algorithm 5 independent bit mutation:

```c
int indep_bit_mutation(individual *ind, pro)
/* Toggles every bit with probability 'pro'. */
{
    int i;
    for(i=0;i<ind->num_T;i++)
    {
        if (drand(1) <= pro)
            ind->map_string[position] = irand(ind->num_P);
    }
    /* evaluate new individual */
    re_sch(ind);
    ind->mcm = eval_mcm(ind);
    return (0);
}
```

4.3 Enhancement to the Search Procedure

We present an enhancement to GA for leading the search to the feasible area by eliminating the infeasible area before the search starts.
Figure 11 Example to illustrate the need to check actor ordering constraints before GA

Considering the HSDF graph in Figure 11, where actors A, B, C, D have fixed precedence constraints inherited from the PIM, regardless of the actual mapping and scheduling:

\[ A < B, A < D, B < D, A < C \]

where ‘A < B’ means that A precedes B. All such orderings can be obtained from computing the transitive closure of the HSDF graph using the Floyd-Warshall algorithm. We can use these ordering constraints to reduce the search space by setting the corresponding \( a_{ij} \) value in the order encoding matrix to 11. This ensures that these bits \( (a_{ij}) \) will be held constant during the GA workflow, hence the search space is reduced. When the number of edges in the HSDF graph is large, these kinds of pre-defined orderings will help prune a large portion of the infeasible ordering space in GA.
5.1 Experiment setup

The experiments are run on a Linux workstation with an Intel dual-core 2.4 GHZ 64-bit processor and 4GB of main memory.

We use the software tool SDF3 [9] to generate random HSDF graphs. The WCET of each application actor ranges from 10 to 50. The network actors have WCET set to 5. Buffer size of each edge ranges from 1 to 5 tokens; the probability of an edge have initial tokens is 0.2. We use the PISA software package to implement the GA.

We use different mutation and crossover strategies in GA in the experiment: one point crossover and uniform crossover; independent mutation and one bit mutation. Maximum generation of the population is set to 20. The mutation probability is set to 50%. Recombination probability is 80%.

5.2 Experiment results

We compare the performance and scalability of the stochastic search techniques with the exact solution technique based on SAT solving.
<table>
<thead>
<tr>
<th>Actors</th>
<th>MCM(sat)</th>
<th>MCM(GA)</th>
<th>Time(sat)</th>
<th>Time(GA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>30</td>
<td>8</td>
<td>156</td>
</tr>
<tr>
<td>4</td>
<td>35</td>
<td>35</td>
<td>8</td>
<td>244</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>50</td>
<td>16</td>
<td>288</td>
</tr>
<tr>
<td>6</td>
<td>116</td>
<td>116</td>
<td>32</td>
<td>372</td>
</tr>
<tr>
<td>7</td>
<td>151</td>
<td>151</td>
<td>632</td>
<td>400</td>
</tr>
<tr>
<td>10</td>
<td>213</td>
<td>278</td>
<td>2480</td>
<td>552</td>
</tr>
<tr>
<td>11</td>
<td>227</td>
<td>261</td>
<td>1510</td>
<td>802</td>
</tr>
<tr>
<td>12</td>
<td>162</td>
<td>210</td>
<td>5630</td>
<td>1120</td>
</tr>
<tr>
<td>15</td>
<td>235</td>
<td>368</td>
<td>184500</td>
<td>1650</td>
</tr>
<tr>
<td>20</td>
<td>316</td>
<td>447</td>
<td>600000+</td>
<td>2400</td>
</tr>
</tbody>
</table>

Table 2 Experiment results

Figure 12 Comparison of the MCM values obtained with GA and SAT solver
Figure 13 Comparison of Algorithm running times

Table 2 lists results of the experiment. The number of actors range from 2 to 20, and the number of processors is 2.

The SAT solver technique returns the exact results of the problem: the mapping and scheduling with the best (minimum) MCM. As shown in Figure 12, when the number of actors is relatively small (below 10), GA can return the optimal solution since the search space is small. With a larger number of actors, Figure 12 shows that GA can no longer find the optimal solution when the search space is larger; hence there is a larger chance of missing the optimal solution using GA.

The SAT solving approach is limited by the algorithm running time. Figure 13 shows that the algorithm running time increases at an exponential rate with the number of actors. When the number of actors is 20, running time of the SAT solving approach goes above 10 minutes, which is almost 300 times the cost of GA approach.

We also test the case when the number of actors is very large. The result shows that when the number of actors increased to 40, sometimes GA can not find a feasible
solution. We explain the reason in the discussion.

## 5.3 Discussions

Algorithm running time depends on the search space size, which is influenced by many factors like: number of actors, number of processors, buffer size limitation, position and number of the initial tokens, the WCET of actors, and the shape and size of the HSDF. A larger searching space makes it more likely for GA to miss good solutions.

One property that influences the efficiency of GA technique comparing to Symbolic Techniques like SAT solvers technique is that: when the number of feasible solutions in the exploration search space is extremely low (compared to the size of the search space). It will be difficult for GA to find any feasible solution, since most of the individuals will contain a deadlocked state.

During the evolutionary procedure of GA, the algorithm tries to find good genes, and keep them in the population. This is based on the comparison of two individuals: the algorithm assumes that when an individual has a better MCM than the other one, it must have some good genes in it. Then the individuals with good genes will have a better chance to recombine with other individual in order to pass its gene to the next generation. In this way, after generations of evolution, it is expected that good genes are accumulated in the population, which will make the individuals have good performance.

So when it comes to a case that the search space is very large and at the same time the number of feasible solutions is extremely small (this could happen when a HSDF graph can easily cause deadlock when applying mapping and scheduling). It will be difficult for GA to find a feasible solution during the first generation. Then, it will
not be able to find good genes because all the individuals are considered as equal. In this case, the GA actually returns to doing random search in the area, trying to find a feasible individual.

This explains the results in the experiment when the number of actors is large (set to 40 in the experiment). In some cases, GA cannot return a feasible solution after 50 generations.

As a conclusion, the experiment results shows that the GA approach can find an acceptably good solution in a fixed amount of time, rather than the best possible solution. It has better scalability than the SAT approach.
CHAPER 6

CONCLUSIONS

In this thesis, we have addressed the problem of mapping a Timed Homogeneous Synchronous Dataflow (HSDF) onto a multiprocessor platform, where multiple processors are connected with a communication substrate with guaranteed latency, e.g. a hard real-time Network-on-Chip, with the object of maximizing total throughput.

The technique presented is based on Genetic Algorithms. We use GA to explore the search space: generate feasible actor-to-processor mapping and scheduling of tasks on processor, then use graph theoretic techniques to calculate application throughput for each mapping and scheduling strategy in the quest for the optimal strategy with maximum throughput. We compare the result with the SAT-based technique, which guarantees to return the optimal solution. Experiment results indicate that GA can provide relatively good mapping and scheduling strategies when the search space is too large to handle using SAT-Based techniques.
REFERENCES

[1] Weichen Liu, Mingxuan Yuan, Xiuqiang He, Zonghua Gu, and Xue Liu  
Efficient SAT-Based Mapping and Scheduling of Homogeneous Synchronous  
Dataflow Graphs for Throughput Optimization, Proceedings of the 2008 Real-Time  
Systems Symposium  
optimization approach for statically scheduling task graphs with communication  
delays to multiprocessors. in DATE, R. Lauwereins and J. Madsen, Eds. ACM, 2007,  
pp. 57–62.  
Task Allocation Problem in Distributed Architectures. in RTSS. IEEE Computer  
Society, 2006, pp. 147–158.  
and throughput constraints for synchronous dataflow graphs, in DAC, E. Sentovich,  
allocation for throughput-constrained synchronous dataflow graphs, in DAC, 2007,  
pp. 777–782.  
hard-real-time jobs on a heterogeneous multiprocessor, in EMSOFT, C. M. Kirsch


