國立交通大學

資訊科學與工程研究所

碩 士 論 文

多 重 配 送 處 理 器 架 構 下 的 延 伸 指 令 集 探 索

Instruction Set Extension Exploration in Multiple-issue Architectures

研 究 生：陳志遠

指導教授：單智君　教授

中 華 民 國　九 十 六　年 八 月

多 重 配 送 處 理 器 架 構 下 的 延 伸 指 令 集 探 索

Instruction Set Extension Exploration in Multiple-issue Architectures

研 究 生：陳志遠　　　　　　Student：Chih-Yuan Chen

指導教授：單智君　　　　　　Advisor：Jyh-Jiun Shann

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

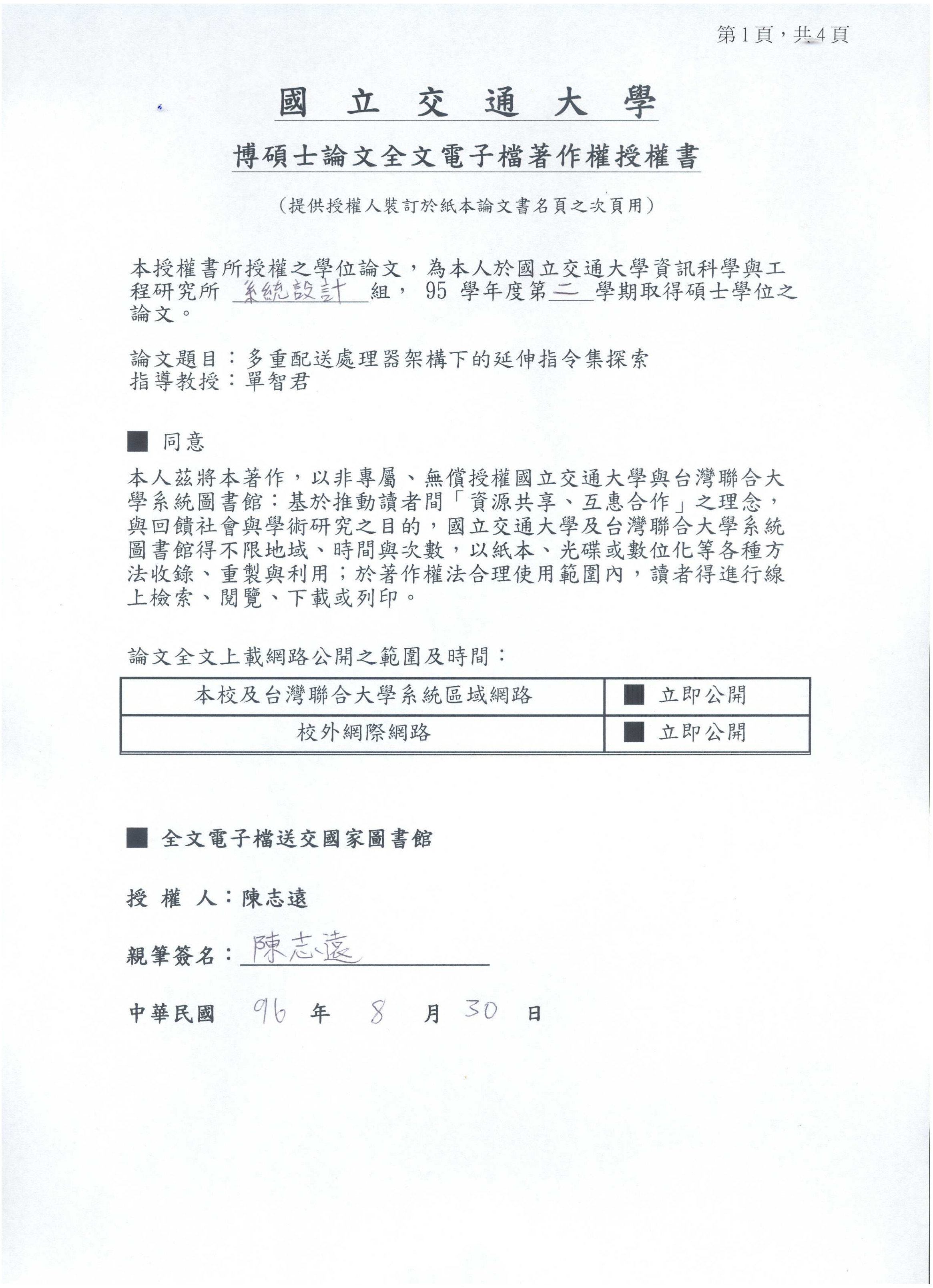
in

Computer Science

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十六年八月



國 立 交 通 大 學

博碩士論文全文電子檔著作權授權書

（提供授權人裝訂於紙本論文書名頁之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程研究所 系統設計 組， 95 學年度第 二 學期取得碩士學位之論文。

論文題目：多重配送處理器架構下的延伸指令集探索
指導教授：單智君

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間：

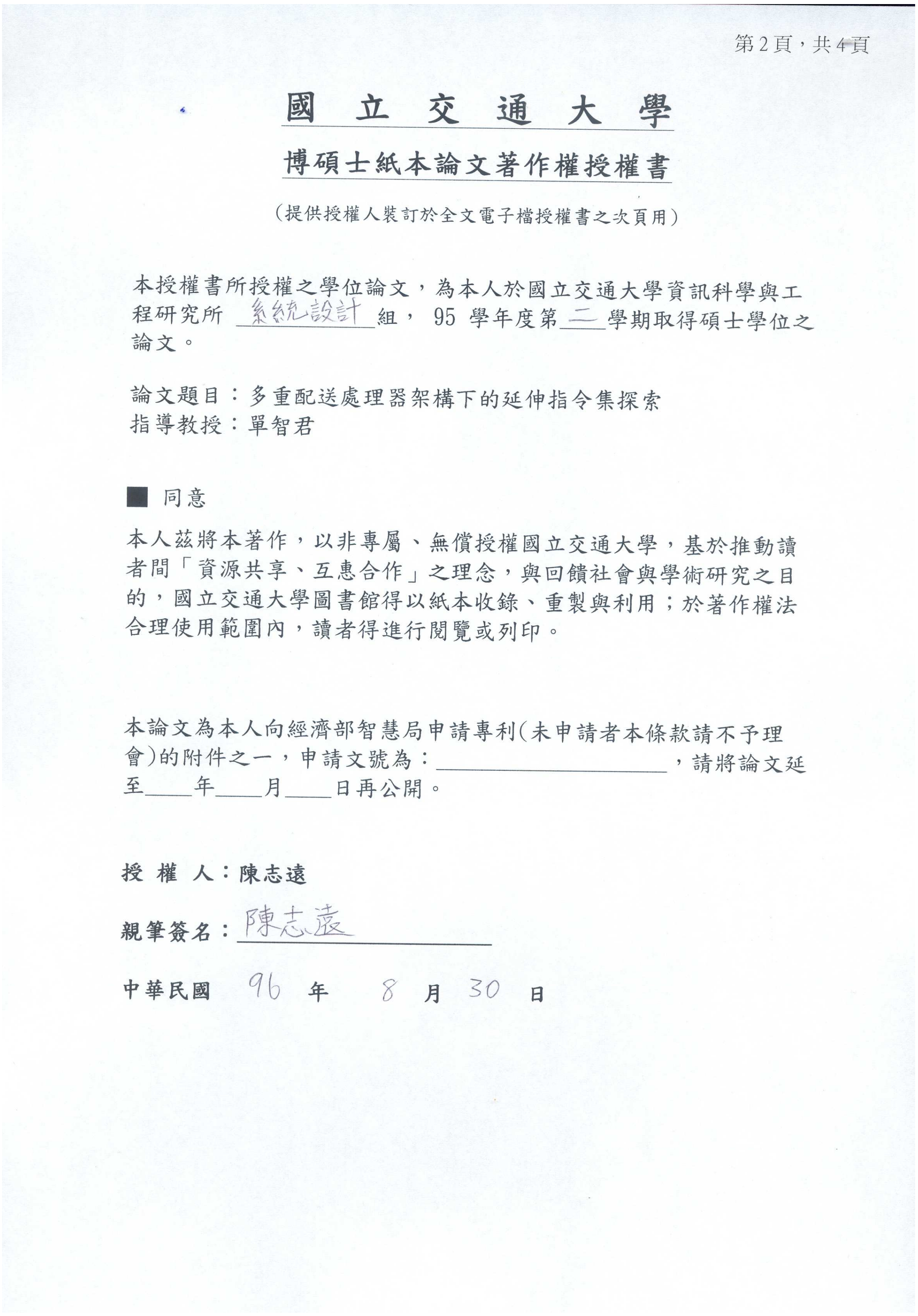
本校及台灣聯合大學系統區域網路	■ 立即公開
校外網際網路	■ 立即公開

■ **全文電子檔送交國家圖書館**

授 權 人：陳志遠

親筆簽名：陳志遠

中華民國 96 **年** 8 **月** 30 **日**



國 立 交 通 大 學

博碩士紙本論文著作權授權書

（提供授權人裝訂於全文電子檔授權書之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程研究所 系統設計 組， 95 學年度第 二 學期取得碩士學位之論文。

論文題目：多重配送處理器架構下的延伸指令集探索
指導教授：單智君

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：____________________，請將論文延至____年____月____日再公開。

授 權 人：陳志遠

親筆簽名： 陳志遠

中華民國 96 **年** 8 **月** 30 **日**

國立交通大學
研究所碩士班

論文口試委員會審定書

本校 資訊科學與工程 研究所 陳志遠 君

所提論文:

多重配送處理器架構下的延伸指令集探索

Instruction Set Extension Exploration in Multiple-Issue Architectures

合於碩士資格水準、業經本委員會評審認可。

口試委員： 謝萬雲 盧能彬

鍾崇斌 單智君

指導教授： 單智君

所　　長： [illegible]

中 華 民 國 九十六 年 七 月 二十五 日

多 重 分 配 架 構 下 的 延 伸 指 令 集 探 索

學生：陳志遠　　　　　　　　　　　指導教授：單智君　教授

國立交通大學資訊科學與工程研究所 碩士班

摘要

為了滿足現代嵌入式裝置高效能的需求，現代的嵌入式處理器提供了延伸指令集(ISE)供設計者定義，或是增加指令的配送寬度。通常來說這兩種方法被視為是不同的，假使我們能結合兩種方法: 執行延伸指令集並讓指令同時執行，就可以節省更多的執行時間。然而大多數的延伸指令集探索演算法，並不大適用於多發射架構下，那是由於缺乏兩個重要的考量:(1) 在多發射處理器的架構下，並不是所有指令都在關鍵路徑上，如果將不是在關鍵路徑上的指令包成 ISE,那便會浪費額外的面積 (2) 在多發射處理器架構下，產生一道新的 ISE 後，關鍵路徑可能會改變。為了要滿足這些考量，我們提出了一個 ISE Exploration 的演算法。實驗結果顯示，在多發射處理器的環境下，使用一道 ISE 和不使用 ISE 相比，我們的方法可以達到 17.17%, 12.9% 和 14.79%(最大，最小和平均) 的執行時間縮減。再者，我們的方法在相同的面積限制下和之前的研究相比，提高了 11.39%,2.87%和 7.16%(最大，最小和平均)的執行速度。

Instruction Set Extension Exploration in Multiple-Issue Architectures

student：Chih-Yuan Chen Advisors：Jyh-Jiun Shann

Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

Abstract

To satisfy high-performance computing demands in modern embedded devices, current embedded processors architectures either provide designer with possibility to define instruction set extension (ISE) or to increase instruction issue width. In general, both approaches are regarded as different; if we can integrate both approaches to execute ISE(s) and original instruction(s) in parallel, then further execution time can be saved. Most ISE exploration algorithms, however, are unlikely to be used in the multiple-issue processor due to the lack of two important considerations: (1) for multiple-issue processor, not all operations locate on the critical path; if operations locating on the non-critical path are grouped as ISE, then it results in unnecessary waste of silicon area; (2) the critical path may change after generating a new ISE in multiple-issue processor. To conform to these considerations, in this paper, we propose an ISE exploration algorithm for multiple-issue processor.

誌謝

首先感謝我的指導老師 單智君教授，在他的諄諄教誨、辛勤指導與勉勵下，得以順利完成此篇論文。同時感謝我的口試委員謝萬雲、盧能彬、鍾崇斌，以及單智君教授，在他們的建議之下，使此篇論文更加完整。

感謝博士班學長—吳奕緯學長，以及其他的博士班學長。也感謝實驗室其他同學們熱心的與我討論，給我意見和鼓勵。

此外，感謝諸位同學和學弟妹們，你們的陪伴讓我的生活充滿歡樂；也讓這兩年來的研究生活更加的多采多姿與充實。最後感謝我的家人，謝謝你們在背後全心全意的支持我、關懷我與鼓勵我。讓我在這研究的路上走得更順利，進而能更全無後顧的用功學習。

所有支持我、勉勵我的師長與親友，奉上我最誠摯的感謝與祝福，謝謝你們。

陳志遠
2007.8.27

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Instruction Set Extension

Recently, more and more applications are dramatically driving up the performance demands on embedded system design. Instruction set extension (ISE) is an effective way to meet the growing efficiency demands for both circuit and speed in embedded applications. Since several instruction patterns are executed frequently in most applications, grouping these instruction patterns into the ISEs is an effective way of improving the performance. ISEs are realized by using application specific functional units (ASFU) within the execution stage of pipeline.

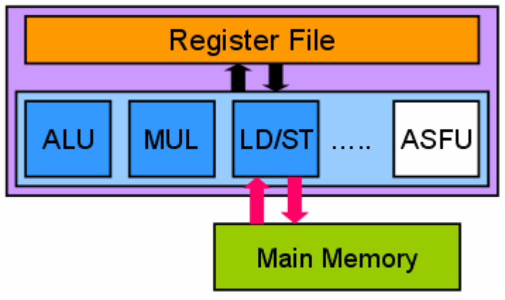


Figure 1.1.1: The diagram of CPU core and ASFU

1.2 Physical Constraints

Instruction Set Architecture (ISA) Format

ISA format usually imposes two kinds of constraints on ISEs. The first is the input/output register number of ISEs. This is due to instruction format limitation or number of register file read/write ports. The other constraint is the number of ISEs. Generally speaking, the number of ISEs can’t exceed number of unused opcode.

Total Silicon Area

The total silicon area restricts extra area used by ASUF.

1.3 Why ISE in Multi-Issue Architecture

Recently, next-generation digital entertainment and mobile communication devices are driving the need for high-performance processing solutions. In order to satisfy this demand, current embedded processors architectures either provide designer with possibility to define customized instruction set extension (ISE) [4, 5, 6, 7, 8, 9, 10, 11, 12 and 13] or increase instruction issue width [2 and 3].

Using ISE and increasing issue width are usually considered as different approaches to accelerate application(s) execution. Since several instruction patterns are executed frequently in most applications, grouping these instruction patterns as new instruction, i.e. instruction in ISE, and realizing this new instruction on the application specific functional units (ASFU) would have benefit in execution time reduction. For the sake of simplicity, we call instruction(s) in ISE as ISE(s) hereafter. On the other hand, extending issue width increases the opportunity of executing instructions in parallel. At this point, the questions have emerged: is there opportunity to reduce the execution time by combining both approaches, i.e. by deploying ISE in multiple-issue processor?

The answer is yes. Because even the issue width and hardware resources are infinite, performance is still severely limited by data dependency. For example, consider the DFG shown in figure 1. If the issue width and hardware resources are infinite, this DFG still spends at least four cycles to execute. However, using ISE is to compress the execution time of operations which locate on the critical path. That is, increasing issue width cannot reduce the execution time of operations having data dependency, but using ISE can. Hence, using ISE can be considered as complementary approach for increasing issue width to reduce execution time. Fig. 1, in which C-*N* represents *N*-th cycle (e.g. C1 is first cycle), illustrates this argument. Note that in Fig. 1, we do not take register read/write port constraint into account, and ISE cannot directly access memory (i.e., no load/store instructions are packed into ISE). Extending issue width lets independent operations execute simultaneously (e.g. comparing single-issue with 2-issue in Fig. 1). On the other hand, using ISE is to pack operations which locate on the critical path into a new instruction (e.g. comparing without ISE to with ISE in Fig. 1) and to execute this new instruction in a fast hardware (i.e. ASFU). Therefore, if we combine these two approaches, i.e. deploying ISE in multiple-issue processor, then more performance improvement can be achieved.

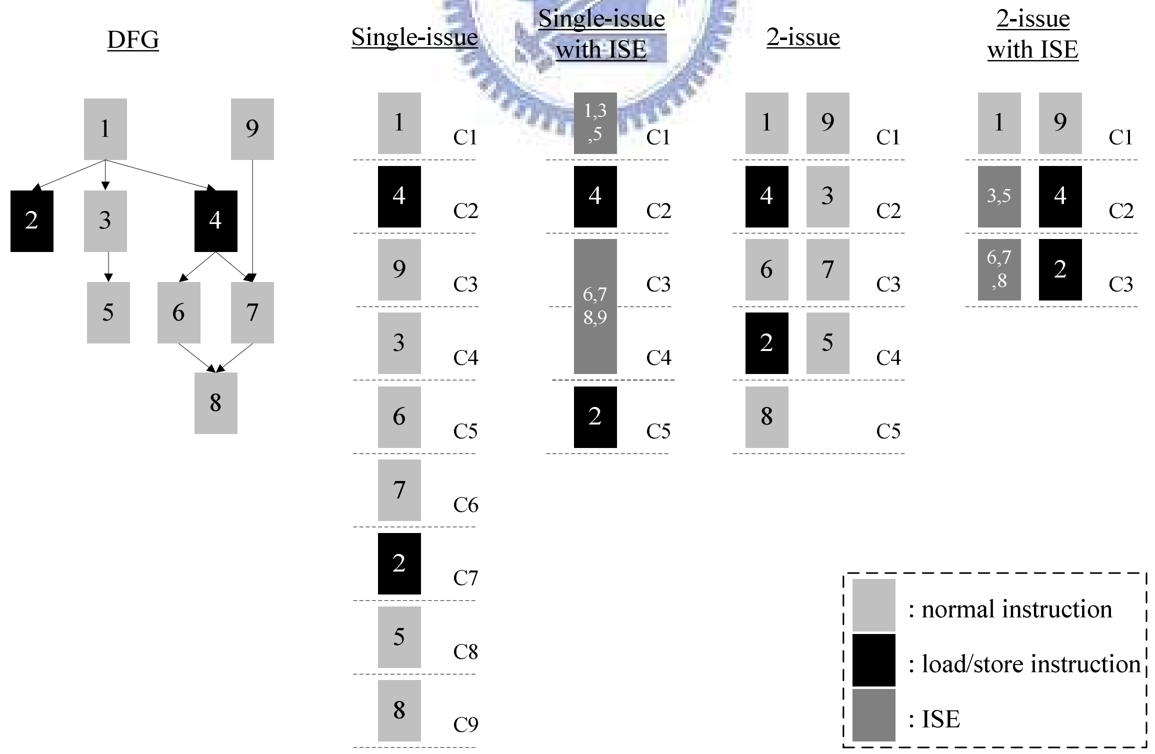


Figure 1.3.1: ISE exploring results for different architectures

1.4 Why New ISE Exploration algorithm for Multi-Issue Architecture

Current ISE exploration algorithms [4, 5, 6, 7, 8 and 13] only consider the legality of operations, but do not consider the location of operations. (A legal operation means that after encapsulating this operation into ISE *X*, ISE *X* will not violate any predefined constraints.) In order to reduce execution time, ISE exploration must pack operations locating on the critical path into ISE(s). In other words, encapsulating operations which locate on the non-critical path into ISE(s) hardly gains any performance improvement and wastes silicon area. However, current ISE exploration algorithms overlook this point. Current ISE exploration algorithms are, therefore, unlikely to be used in the multiple-issue processor. To illustrate this argument, we schedule two results shown in Fig. 1.3.1. First one is to schedule the result of "single-issue with ISE" on a 2-issue processor (case 1), and second one is also to schedule the result of "2-issue with ISE" on a 2-issue processor (case 2). Obviously, case 2 has shorter execution cycle and consumes less silicon area than case 1. It clearly demonstrates the benefit of considering the location of operations. In addition, in multiple-issue processor, the critical path may change after generating a new ISE. This causes that instructions must be rescheduled after generating a new ISE to identify the critical path again. To summarize the above two points, identifying the critical path is essential for exploring ISE in multiple-issue processor. That is, designing an ISE exploration algorithm for multiple-issue processor must consider instruction scheduling.

1.5 Motivation

Instruction set extension could improve processor performance efficiently. However, current researches for ISE exploration algorithms consider only single issue architecture processor. For multiple-issue architecture, only composing the operations located on the critical path as ISE can reduce execution time.

1.6 Objective

Design an ISE Exploration algorithm by consider the operations in critical path to generate ISEs to reduce execution time in multiple-issue architectures.

Chapter 2

Relative Works and Background

2.1 Relative Works

ISE design flow comprises application profiling, basic block selection, ISE (candidate) exploration, ISE (candidate) merging, ISE selection as well as hardware sharing, and ISE replacement. After application profiling, basic block(s) is selected as the input of ISE exploration based on their execution time. ISE exploration explores legal instruction pattern as ISE candidate, which have to conform to predefined constraints [4, 5, 6, 7, 8 and 13], e.g. pipestage timing, instruction set architecture (ISA) format, silicon area and register file. In ISE merging stage, the algorithm merges the ISE *B* into ISE *A*, if ISE *B* is a subgraph of ISE *A*. After executing ISE merging, ISE selection chooses as many ISEs as possible to attain the highest performance improvement under predefined constraints [9, 10, 11, 12 and 13], such as silicon area and ISA format. To achieve higher hardware utilization, hardware sharing is also performed at this stage (ISE selection). Strictly speaking, the results of both ISE (candidate) exploration and ISE (candidate) merging are ISE candidate(s). But for the sake of simplicity, ISE candidate is sometimes called ISE. In addition, because we only focus on ISE exploration in this paper, the algorithms of other steps do not be addressed, and these can be referred in the [8, 9, 10, 11, 12 and 13].

Pozzi [4] proposed an algorithm to examine all possible ISE candidates such that it can obtain an optimal solution. This maps the ISE search space, such as a basic block, to a binary tree, and then discards some portion of the tree that violates predefined constraints. Nevertheless, this algorithm is highly computing-intensive, so does not process a larger search space. For instance, if a basic block has N operations, and each operation has only one hardware implementation option, then it has 2^N possible ISE patterns (legal or illegal). Notably, one ISE candidate may consists of one or multiple legal ISE pattern(s). When $N = 100$ (the standard case), then the number of possible ISE patterns is 2^{100}. Obviously, this number of patterns cannot be computed in a reasonable time. To decrease the computing complexity, heuristic algorithms derived from genetic algorithm [4], Kernighan-Lin (KL) [5], greedy-like algorithm [6] and ant colony optimization algorithm [8] have been developed. An Integer Linear Programming formulation of the ISE exploration was presented in [7]: in this case, the enumeration of subgraphs is implicit in the formulation’s constraints, and the worst-case complexity is still exponential. Nevertheless, all algorithms [4, 5, 6, 7 and 8]only consider the legality of operations when exploring ISE

2.2 Background — Ant Colony Optimization (ACO) Algorithm

Why Ant Colony Optimization Algorithm？

In order to indicate which part of a DFG is going to be ISE; the implementation of nodes should be decided. If we only consider the situation that there is only single hardware implementation option of a node, then there will be 2^N possible ISE patterns (legal or illegal) that N is the DFG size. When N is 100 (it’s a usually case), the combinations is emphatic 2^{100}！Obviously, this is a NP-hard problem. For the sake of an efficiently solution, the way of evolutionary computation which is operative to many existing NP-hard problems is considered.

There are many computation models belong to evolutionary computation, like genetic, simulated annealing, etc. One of them named “Ant Colony Optimization” is thought to be the easiest one to map to the problem. The selection among the models is processed by the difficulty of the mapping to the problem. An intuitive and easier mapping usually brings a simple and effective design of the algorithm.

One of the concepts of ACO is the selection a path among many choices (one or two or more) to get the shortest path. I think the selection among many different implementation options of each node is just like that. This is the main reason that ACO outperforms other models. The only problem is how do the nodes "communicate" to each other. The merit computation in the design takes it into account.

Basic Idea of Ant Colony Optimization Algorithm

Ant Colony Optimization algorithm [1 and 2] is inspired by the behavior of ants in finding paths from the colony to food and has been extensively used to solve many optimization problems. Initially, ants wander randomly and lay down pheromone on the paths have been passed through. The density of the pheromone determines the probability of which path the next ant will pass through. Since the pheromone evaporates with the time, a shortest path gets marched over faster and thus has the higher density of pheromone. After a period of time, i.e. several iterations, more and more ants choose the shortest path such that the density of pheromone on this path grows increasingly. Finally, each ant almost chooses the shortest path and the pheromones of other paths evaporate to nearly zero.

Figure 2.2.1 is an example. Suppose 50 ants are in the ant colony. Now they are going to find food. There are two paths to get food. One is twice longer than the other. At t = 1, there is no pheromone on both paths. The ants choose paths with equal probability. Suppose 25 ants choose one path, and 25 ants choose another. One ant leaves one unit of pheromone on the path. But the pheromone evaporates 5 units after t = 1. So the paths ant passed has 25 – 5 = 20 pheromone. At t = 2, ants start again. After t = 2, we can see the pheromone on each path segment. Next time, the right hand side path will be chosen by ants with higher probability than the left hand side path.

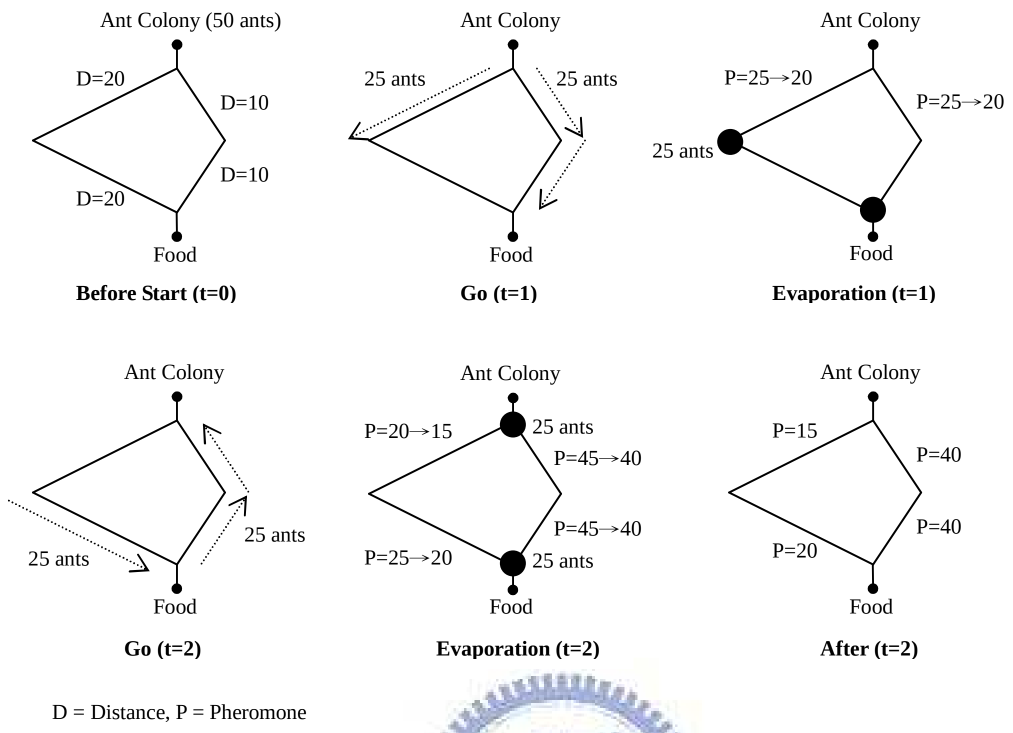


Figure 2.2.1: An example of ant behavior

Chapter 3

ISE Exploration

In this paper, the purpose of ISE exploration is to find frequently executed instruction patterns as ISE candidates and evaluates all implementation options of each operation in ISE candidates to minimize the execution time with less silicon area. The input and output of ISE exploration algorithm are BBs and ISE candidates as well as their implementation option, respectively. Implementation option(s) of an operation represents its implementation method(s), and can be roughly divided into two categories, hardware and software.

The flow of ISE exploration is briefly described as follows: each input BB is first transformed to data flow graphs (DFG), and an implementation option (IO) table which represents all implementation options for an operation is appended to each operation in DFG. In this extended DFG, ISE exploration algorithm is repeatedly executed until no ISE candidate can be found. Note that ISE exploration algorithm only explores one ISE candidate at each round. A round usually consists of multiple iterations. Initially, ISE exploration algorithm chooses one implementation option in each operation according to a probability value (*p*). The probability value (*p*) is a function of pheromone and merit values. The meaning of pheromone is the same with the pheromone in the ACO algorithm, i.e. how many times an implementation option is chosen in previous iterations. The merit value represents the benefit of one implementation option being chosen. After making a choice, the pheromone value is updated. And then, the algorithm evaluates implementation option of each operation in DFG, i.e. calculates their merit value, according to which implementation option is chosen in its neighboring ones at previous iteration. Above process are iteratively performed until the probability values (p) of all operations in DFG have exceeded a predefined threshold value, P_END.

3.1 ISE Design flow

The ISE design flow, as illustrated in Figure 3.3.1, comprises application profiling, basic block selection, ISE (candidate) exploration, ISE (candidate) merging, ISE selection and hardware sharing as well as ISE replacement and instruction scheduling. After application profiling, basic block(s) is selected as the input of ISE exploration based on their execution time. ISE exploration explores legal instruction pattern as ISE candidate, which have to conform to predefined constraints [4, 5, 6, 7, 8 and 13], e.g. pipestage timing, instruction set architecture (ISA) format, silicon area and register file. If only one ISE is explored, then the algorithm directly enters final stage (ISE replacement and instruction scheduling); otherwise, the algorithm goes to next stage (ISE merging). In ISE merging stage, the algorithm merges the ISE B into ISE A, if ISE B is a subgraph of ISE A. To avoid unnecessary performance degradation, the merging process is performed if the following conditions are satisfied: (1) the execution cycle of ISE B is equal or larger than that of the identical subgraph (identical to ISE B) in ISE A, and (2) ISE A and ISE B do not be executed simultaneously. After generating ISE candidates, ISE selection chooses as many ISEs as possible to attain the highest performance improvement under predefined constraints [9, 10, 11, 12 and 13], such as silicon area and ISA format. To achieve higher hardware utilization, hardware sharing is also performed at this stage. Hardware sharing is the assignment of a hardware resource to more than one operation within different ASFUs. Same with ISE merging, hardware sharing also follows the same rules as described above to avoid performance degradation. Finally, the ISE replacement is performed to discover all instruction patterns (i.e. subgraphs) in the DFG that match selected ISEs, prioritizes these matches and replaces the matches with ISEs. Strictly speaking, the results of both ISE candidate exploration and ISE candidate merging are ISE candidate(s). But for the sake of simplicity, ISE candidate is usually called ISE. Hence, in this paper, we use ISE to replace ISE candidate. In addition, because we only focus on ISE exploration in this paper, the algorithms of other steps would not be addressed, and these can be referred in the [8, 9, 10, 11, 12 and 13].

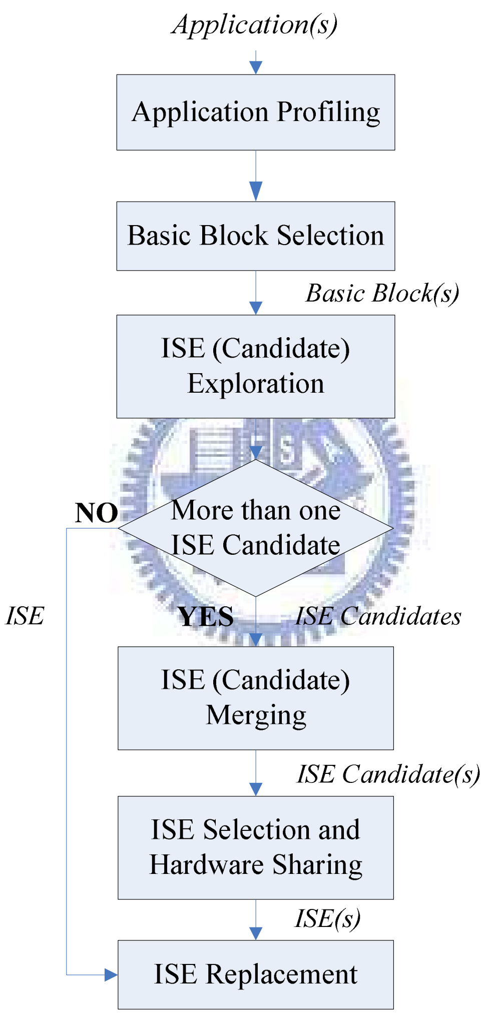


Figure 3.1.1: ISE design flow

3.2 How to apply ACO algorithm to ISE exploration

ISE exploration in multiple-issue processor is to choose an implementation option for each operation and determine the execution order of operation. Exploring ISE in a DFG can be viewed as a search in the space of possible or feasible solutions. Here, the solution means a set of ISE candidate found in a DFG. To apply ACO algorithm, the search space is organized as a search tree. A path from root to leaf in the search tree is considered as one of possible or feasible solutions. After constructing the search tree, we place ant colony and food at root and leaf of search tree, respectively, and let ants make decision (choose an implementation option, and select one succeeding operation if need) level by level to construct the solutions. Selecting the shortest path from ant colony to food can be viewed as similar to choosing the best implementation option (hardware or software) and determining the optimal execution order for all operations.

Figure 4 is an example to illustrate above concept. The leaf hand of Fig. 4 shows the dependence of O1, O2 and O3, the search tree is depicted at the right hand of Fig. 4. In this example, we assume that there are three operations, namely O1, O2 and O3, and each operation has two hardware (H1 and H2) and two software (S1 and S2) implementation options. Since the possible execution order for operation O1, O2 and O3 are O1→O2→03 and O1→O3→02, respectively, there exist two paths after choosing one implementation option at O1.

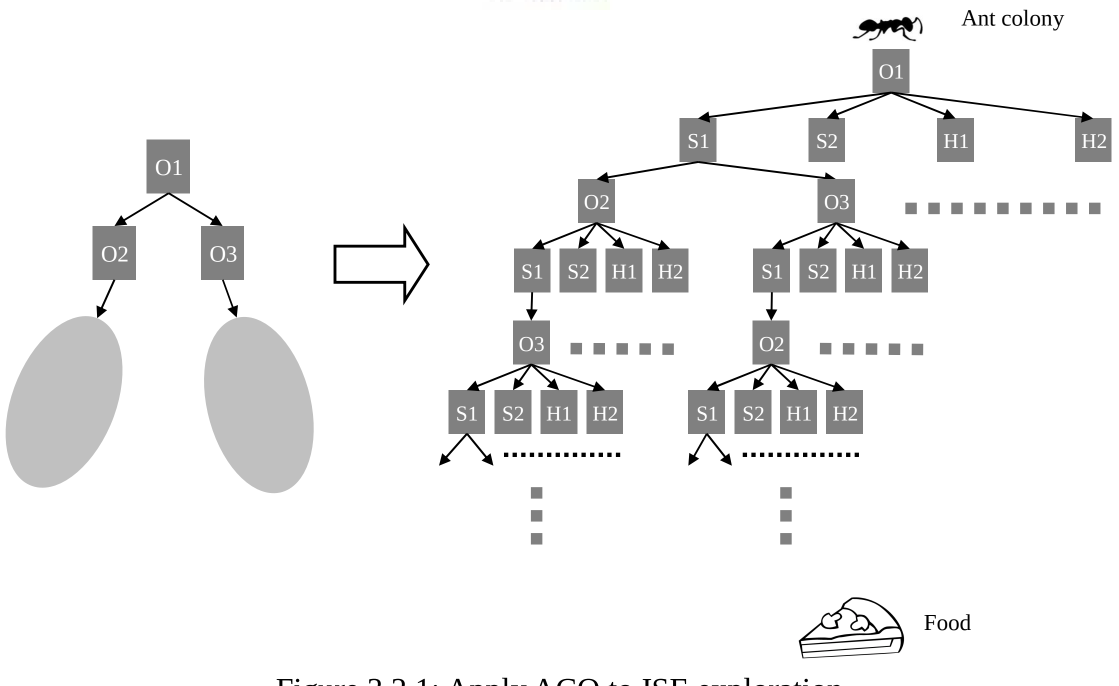


Figure 3.2.1: Apply ACO to ISE exploration

Chapter 4 ISE Exploration in Multiple-Issue Architecture

The input and output of ISE exploration algorithm are selected basic block(s) and ISE candidate(s) as well as its (their) hardware implementation options, respectively. Figure 4.0.1 is an example. Before exploring ISE, a basic block must be transformed to a data flow graph (DFG). DFG is represented by a directed acyclic graph $G(V,E)$ where V denotes a set of vertices, and E represents a set of directed edges. Every vertex $v \in V$ is an assembly instruction, called an “operation” or “node” hereafter in basic block. Each edge $(u,v) \in E$ from operation u to operation v signifies that the execution of operation v needs the data generated by operation u.

ISE exploration aims to determine which implementation option should be used by which operation. As mentioned early, if the operations locating on the non-critical path are packed into ISE, then there does not only not improve performance, but also waste silicon area. To avoid this situation, the algorithm must identify which operation locates on the critical path before starting to encapsulate operations into ISE.

Exploring ISE in multiple-issue architecture is to assign each operation in DFG a time slot and an implementation option such that execution time is minimal, and under that, consumes less silicon area. In Fig. 4.0.1, we assume that the issue width of processor is two, and that each operation has only one hardware/software implementation option. After exploring, operation 3 and 5 as well as operation 6, 7 and 8 choose hardware implementation option, while other operations select software one. ISE is a set of connected/reachable operations that all use hardware implementation option. In Fig. 4.0.1, there are two ISEs in which one consists of operation 3 and 5; another one includes operation 6, 7 and 8.

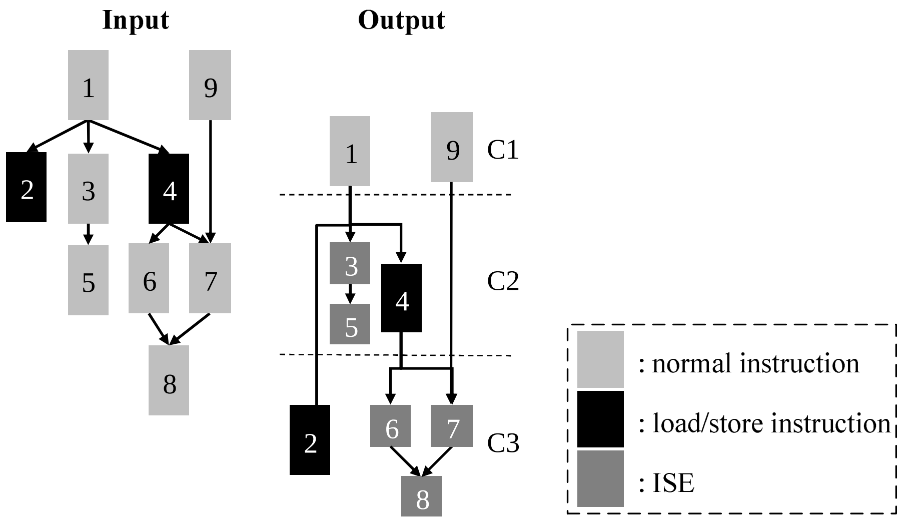


Figure 4.0.1: Example of ISE exploration

The process of the proposed ISE exploration algorithm is to iterate the following steps until no ISEs in a DFG can be explored:

Step 1: Identify the critical path using instruction scheduling and explore ISE to reduce the length of the critical path.

Step 2: Evaluate the result of this iteration and calculate the benefit of all implementation options of operations for next iteration.

To explain this process, an example is depicted in figure 4.0.2. All assumptions are same with Fig.4.0.1. In step 1, the algorithm identifies the critical path (1→4→6→8 and 1→4→7→8) by scheduling instructions, and packs legal operations (6, 7 and 8) into ISE. After generating a new ISE (consists of 6, 7 and 8), all implementation options of operations are evaluated. However, this process is not shown in Fig. 4.0.2. Same with step 1, in step 2, the algorithm also schedules all instructions (including ISE and normal instructions) to distinguish which path is critical, and then encapsulates the operations (3 and 5) locating the critical one into ISE. After that, evaluation process is performed again. In step 3, since no valid operation can be found, the algorithm is terminated. The valid operation means that packing this operation into ISE can have performance gain.

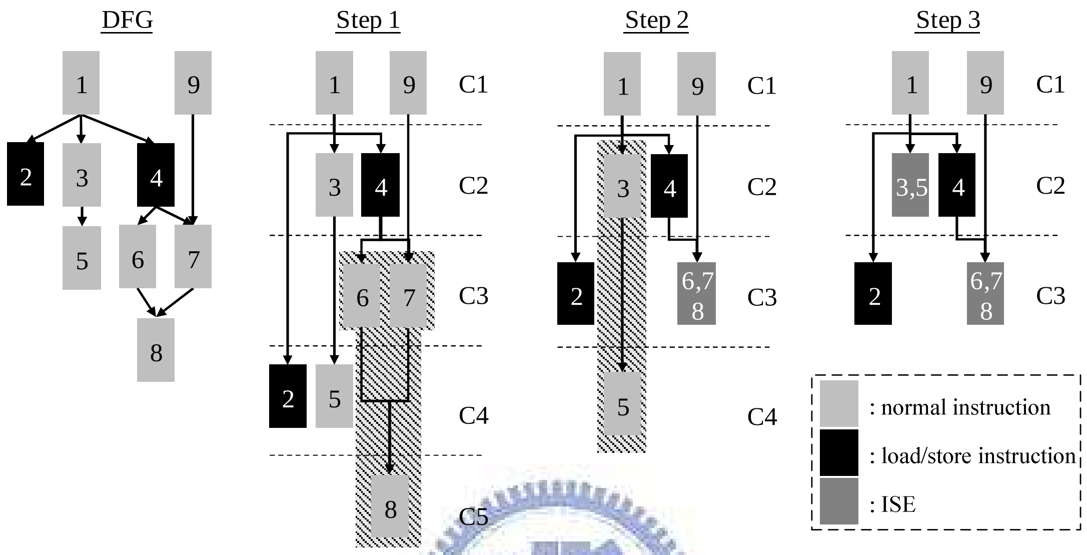
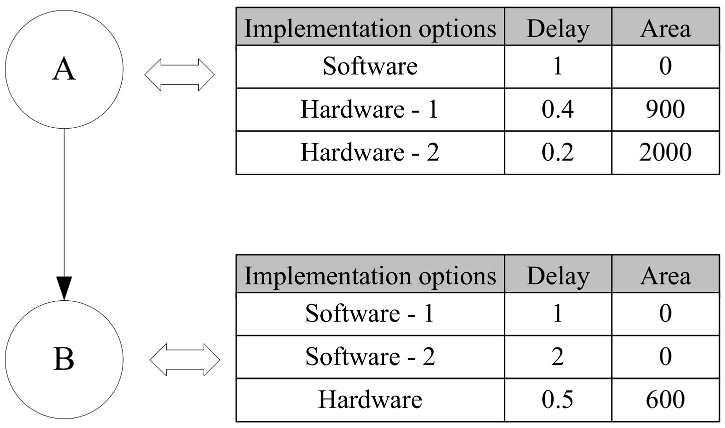


Figure 4.0.2: Example of ISE exploration

4.1 Implementation Option

The implementation option represents the way to execute an operation. An operation usually has multiple implementation options, which can be divided into two categories, namely hardware and software. If an operation is encapsulated into ISE, it means that this operation deploys the hardware implementation option; on the contrary, if not encapsulated, this operation is executed in the processor core. Because of different speed and area requirements, most operations usually have multiple hardware implementation options.

To represent all implementation options for an operation, a table, called implementation option (IO) table, is added to every operation. Each entry in the IO table comprises three fields, namely implementation option, delay and area. The name of implementation option is shown in implementation option field. The delay and area denote the execution time and the extra silicon area cost of one implementation option, respectively. A new graph G^+ is generated after the IO table is added to G. Figure 4.1.1 shows an example of G^+, consisting of two operations, A and B.



Implementation options	Delay	Area
Software	1	0
Hardware - 1	0.4	900
Hardware - 2	0.2	2000

Implementation options	Delay	Area
Software - 1	1	0
Software - 2	2	0
Hardware	0.5	600

Figure 4.1.1: An example of G^+

4.2 Formulation for ISE Exploration

ISE exploration explores ISE candidates in G^+. An ISE candidate in G^+ is a subgraph $S \subseteq G^+$. The proposed ISE exploration can be formulated as follows.

ISE exploration: Considering a graph G^+, obtain subgraph $S \subseteq G^+$, and evaluate the implementation options of vertex $v \in S$ to minimize the execution cycle count while reducing the silicon area as many as possible under the following constraints:

1. $\mathrm{IN}(S) \leq \mathrm{N_{in}}$,

2. $\mathrm{OUT}(S) \leq \mathrm{N_{out}}$,

3. S is convex,

4. Load and store operations $\notin S$.

IN(S) (OUT(S)) is the number of input (output) values used (generated) by a subgraph S (i.e. an ISE). The user-defined values $\mathrm{N_{in}}$ and $\mathrm{N_{out}}$ denote the read and write ports limitations of the register file, respectively. For a feasible instruction scheduling, an ISE must observe the convex constraint that the ISE's output cannot connect to its input via other operations not grouped in subgraph S (i.e. ISE). In other words, if no path exists from a operation $u \in S$ to another operation $v \in S$ involving a operation $w \notin S$, then S is convex. To conform to the limitation of load-store architecture, the load and store operations are forbidden from being grouped into ISE.

4.3 ISE Exploration Algorithm

As mentioned above, the proposed algorithm explores ISE iteratively until no ISEs in a DFG can be found. The algorithm, therefore, would be performed for several rounds (a round comprises all steps in figure 4.3.1); except for last round, each round would produce at least one ISE. The kernel of each round (step 2 to step 9 in Fig. 4.3.1) would be executed repeatedly until convergence is achieved. Executing the steps rounded by
gray rectangle once is called one iteration.

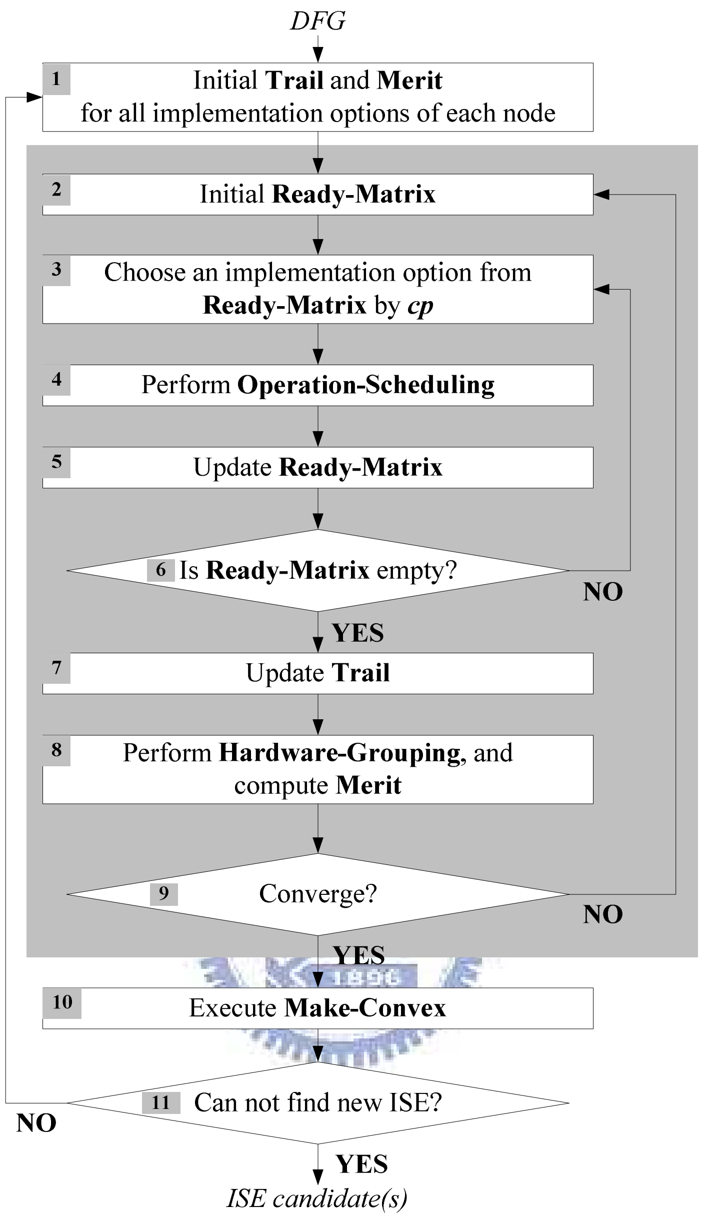

DFG
1 Initial Trail and Merit for all implementation options of each node
2 Initial Ready-Matrix
3 Choose an implementation option from Ready-Matrix by cp
4 Perform Operation-Scheduling
5 Update Ready-Matrix
6 Is Ready-Matrix empty?
NO
YES
7 Update Trail
8 Perform Hardware-Grouping, and compute Merit
9 Converge?
NO
YES
10 Execute Make-Convex
11 Can not find new ISE?
NO
YES
ISE candidate(s)

Figure 4.3.1: ISE exploration flow

At each iteration, the proposed algorithm initially selects one implementation option from Ready-Matrix with respect to a chosen-probability (c*p*), which depends on trail and merit values. Ready-Matrix is a data structure which is very similar with ready list in list scheduling. Figure 4.3.2 is an example of Ready-Matrix; “*” means no this implementation option.

	Operation 1	Operation 2	Operation 3
SW-1	0.04	0.4	0.03
SW-2	0.02	*	0.03
HW-1	0.21	0.04	0.09
HW-2	*	0.04	0.15

Figure 4.3.2: An example of Ready-Matrix

The meaning of trail is the same with the pheromone in the ACO algorithm, i.e. the number of valid chosen times of an implementation option in previous iterations. The valid chosen time is counted only when choosing this implementation option can reduce the execution time. Here, the trail value of hardware and software implementation option *j* of operation *x* is denoted by $trail_{x,HW\text{-}j}$ and $trail_{x,SW\text{-}j}$, respectively. The merit value is defined as the benefit of one implementation option being selected, and it is obtained using the merit function, which is described in detail later. The merit value of of hardware and software implementation option *j* of operation *x* is represented by $merit_{x,HW\text{-}j}$ and $merit_{x,SW\text{-}j}$, respectively. The chosen probability of an operation *x* is derived with:

$$cp = \frac{\alpha \times trail + (1-\alpha)\times merit + \lambda \times \mathrm{SP}}{\sum_{\text{All implementation options in Ready-Matrix}} \{\alpha \times trail + (1-\alpha)\times merit + \lambda \times \mathrm{SP}\}} \tag{1}$$

where α and λ is utilized to determine the relative influence of trail as well as merit and scheduling priority (SP), respectively, and

$$\sum_{\text{All implementation option in Ready-Martix}} cp = 1 \tag{2}$$

The value of SP used in this paper is computed according to the number of child operations; however, this value can also be obtained by other ways, e.g. calculating the mobility of operation. In addition, merit and SP have other meanings. Merit is mainly used to choose one implementation option for operations; while SP is responsible for selecting one operation among all ready ones. (An operation is ready if all dependencies for this operation have been resolved.) Since the difference in merit values between operations may be large, picking an operation to schedule among ready ones is unfair by using such values. To overcome this problem, the merit values of operation must be normalized after performing merit computation (step 8 in Fig. 4.3.1).

After selecting an implementation option, the algorithm schedules the operation which has this chosen implementation option. The scheduling process (Operation-Scheduling) will be described in later. Then, executing following processes to update Ready-Matrix: (1) remove the operation which has the chosen implementation option; and (2) add the operation if all dependencies for this operation have been resolved. The algorithm repeatedly executes step 3 to 6 until all operations are scheduled. After all operations are scheduled, the algorithm updates trail values according to execution time, and then computes merit value of all implementation options of each operation in DFG by using merit function. Each round is repeatedly performed until the end condition is fulfilled, i.e. until converge. The end condition is that for all operations in DFG, the selected-probability (*sp*) of one of implementation options exceeds P_END, which is a predefined threshold value and is very close to 100%. The selected -probability (*sp*) of an operation is derived from:

$$sp = \frac{\alpha \times trail + (1-\alpha)\times merit}{\sum_{\text{All implementation options in one operation}} \alpha \times trail + (1-\alpha)\times merit} \qquad \textbf{(3)}$$

, and

$$\sum_{\text{All implementation option in one operation}} sp = 1 \qquad \textbf{(4)}$$

Noticeably, there are some differences between *sp* (Eq. 3) and *cp* (Eq. 1). The sum in the denominator of Eq. 3 is only over all implementation options in one operation; while, for *cp* (Eq. 1), the sum in the denominator is over all implementation options in Ready-Matrix. A larger P_END has a higher opportunity of obtaining a better result, but typically takes a longer time to converge. An implementation option with the chosen-probability (*sp*) larger than P_END is called a taken implementation option. An ISE is a set of connected/reachable nodes (i.e. operations) all of which have taken hardware implementation option. After convergence, the algorithm executes Make-Convex to let every ISE candidate comply with the convex constraint. But, if an ISE has conformed to the convex constraint, then the algorithm will skip this step. Make-Convex repeatedly divides the ISE candidate that does not conform to the convex constraint into smaller ones until all smaller ISE candidates can comply with convex constraint.

In following paragraph, we describe the several processes/steps used in the proposed algorithm, including Operation-Scheduling, Trail Update, Hardware-Grouping and merit calculation (Merit Function). Here, a DFG is assumed to have k ($k > 0$) operations, each with n ($n > 0$) software implementation option(s) and m ($m > 0$) hardware implementation option(s).

Operation-Scheduling

Operation-Scheduling is used to assign one operation on one time slot under several constrains, including issue-width, number of register read/write ports, number of function units and operation dependency. Assigning an operation using software implementation option is just like statically scheduling instructions in multiple-issue processor. Here, we assume that operation i currently needs to schedule, and the steps of how to schedule it are depicted at figure 4.3.3. In Fig. 4.3.3, LTS_i and CTS_i denote the latest scheduled time slot of parent operations of and current scheduled time slot of operation i, respectively; here, constraints are issue width, number of function units and number of register read/write ports. Note that which function unit (i.e. software implementation option) would be used by operation i has known at previous step.

```
$CTS_i = LTS_i + 1$;
While (violate constraints at $CTS_i$)
    $CTS_i$ ++;
Assign the function unit and the time slot to operation $i$;
Update the resource usage at $CTS_i$;
```

Figure 4.3.3: Operation-Scheduling for software implementation option

To schedule an operation using hardware implementation option is similar with software one, but it still exits differences. The main difference is that it is possible to pack several operations using hardware implementation option in one cycle, but it is impossible to do that for ones using software implementation option. Figure 4.3.4 shows the algorithm of Operation-Scheduling used for hardware implementation option. In Fig. 4.3.4, LP_i presents the parent scheduled at LTS_i; constraints used here are issue width and number of register read/write ports.

```
If ($LP_i$ uses software implementation option)
    $CTS_i = LTS_i + 1$;
    While (violate constraints at $CTS_i$)
        $CTS_i$ ++;
    Assign the time slot to operation $i$;
    Update the resource usage at $CTS_i$;
Else
    $CTS_i = LTS_i$;
    While (cannot pack operation $i$ with other operations into ISE at $CTS_i$)
        $CTS_i$ ++;
    Assign the time slot to operation $i$;
    Update the resource usage at $CTS_i$;
```

Figure 4.3.4: Operation-Scheduling for hardware implementation option

Trail Update

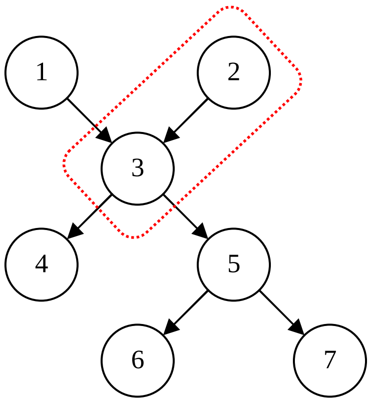
Trail is updated according to the scheduling result of each iteration. The algorithm of trail update is displayed in figure 4.3.5. Here, TET_{new} and TET_{old} are the execution time of current and previous iteration, respectively; ρ_1, ρ_2, ρ_3, ρ_4 and ρ_5 are positive constant values and called evaporating factor as well as very similar to the evaporation rate in ACO. If the execution time is shorter than or equal to previous iteration, it means that the selection of implementation option and the decision of execution order have benefit for execution time reduction. Then, the trail value of the chosen implementation option is raised (increasing ρ_1), a positive constant value, while those of others are reduced (decreasing ρ_2). On the other hand, if the execution time is larger than previous iteration, it means that either or both the selection of implementation option and the decision of execution order are improper. Hence, the trail values of selected implementation option have to be decreased with ρ_3, while those of others are increased with ρ_4. In addition, since the longer execution time may cause by unfit execution order, the all implementation options of the operation, which has higher execution order than previous iteration, are also reduced (subtract ρ_5).

```
If ($TET_{new} \leqq TET_{old}$)
    For software implementation option $i$ ($i$=0 to $n$) of operation $x$ ($x$=1 to $k$) in DFG
        If (the implementation option is selected)
            $trail_{x,SW\text{-}i} = trail_{x,SW\text{-}i} + \rho_1$;
        Else
            $trail_{x,SW\text{-}i} = trail_{x,SW\text{-}i} - \rho_2$;
    For hardware implementation option $j$ ($j$=0 to $m$) of operation $x$ ($x$=1 to $k$) in DFG
        If (the implementation option is selected)
            $trail_{x,HW\text{-}j} = trail_{x,HW\text{-}j} + \rho_1$;
        Else
            $trail_{x,HW\text{-}j} = trail_{x,HW\text{-}j} - \rho_2$;
    $TET_{old} = TET_{new}$;
Else
    For software implementation option $i$ ($i$=0 to $n$) of operation $x$ ($x$=1 to $k$) in DFG
        If (the implementation option is selected)
            $trail_{x,SW\text{-}i} = trail_{x,SW\text{-}i} - \rho_3$;
        Else
            $trail_{x,SW\text{-}i} = trail_{x,SW\text{-}i} + \rho_4$;
        If (execution order of operation $x$ is earlier than previous one)
            $trail_{x,SW\text{-}i} = trail_{x,SW\text{-}i} - \rho_5$;
    For hardware implementation option $j$ ($j$=0 to $m$) of operation $x$ ($x$=1 to $k$) in DFG
        If (the implementation option is selected)
            $trail_{x,HW\text{-}j} = trail_{x,HW\text{-}j} - \rho_3$;
        Else
            $trail_{x,HW\text{-}j} = trail_{x,HW\text{-}j} + \rho_4$;
        If (execution order of operation $x$ is earlier than previous one)
            $trail_{x,HW\text{-}j} = trail_{x,HW\text{-}j} - \rho_5$;
```

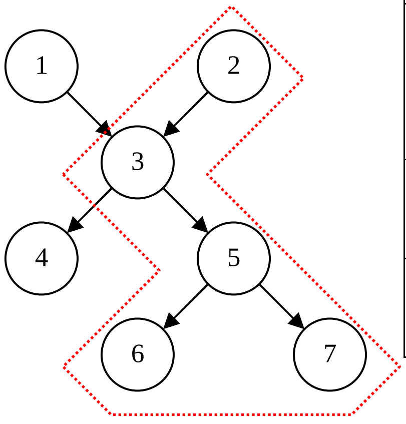
Figure 4.3.5: The algorithm of trail update

Hardware-Grouping

Hardware-Grouping checks whether the operation x can be grouped with its reachable nodes (i.e. operations) as a virtual ISE candidate, and recursively groups operation x with its reachable nodes, which have chosen hardware implementation option in previous iteration, as a virtual ISE candidate, i.e. a virtual subgraph vS_x. The result of Hardware-Grouping of operation x using hardware implementation option j is denoted as $vS_{x,HW\text{-}j}$. *HW-MAX* represents the implementation option having maximal execution time reduction in an operation. Significantly, vS_x is the set of all $vS_{x,HW\text{-}j}$ (i.e. vS_x={ $vS_{x,HW\text{-}j}$ | j = 1 to n}). Using $vS_{x,HW\text{-}j}$, Hardware-Grouping measures the execution time and silicon area of $vS_{x,HW\text{-}j}$. Notably, the execution time of $vS_{x,HW\text{-}j}$ is the critical path time in $vS_{x,HW\text{-}j}$, and the silicon area of $vS_{x,HW\text{-}j}$ is the sum of silicon areas of $vS_{x,HW\text{-}j}$.



Hardware grouping of operation #2



Hardware grouping of operation #5

Operation ID	Choice in previous iteration	Implementation Option	Delay	Area
1	●	**software**	**1**	**0**
2		software	1	0
	●	**hardware**	**0.4**	**600**
3		software	1	0
	●	**hardware**	**0.4**	**600**
4	●	**software**	**1**	**0**
5	●	**software**	**1**	**0**
		hardware 1	0.6	400
		hardware 2	0.3	1000
6		software	1	0
	●	**hardware**	**0.3**	**500**
7		software	1	0
	●	**hardware**	**0.2**	**300**

Figure 4.3.6: Examples of Hardware-Grouping

Figure 4.3.6 depicts the working of the Hardware-Grouping function. The table in Fig. 4.3.6 lists the delay and area of each implementation option of all operations, and specifies the chosen implementation option in the previous selection. In both the top and bottom left of Fig.

4.3.6, nodes grouped by a dotted line are treated as a virtual ISE candidate. For operation #2, Hardware-Grouping groups operation #2 and #3 as a virtual ISE candidate, i.e. vS_2, as shown in the top left of Fig. 11. Because operation #2 only has one hardware implementation option, vS_2 has one evaluation result, namely $vS_{2,1}$ (execution time = 0.8, silicon area = 1200). The bottom left of Fig. 13 is another example, in which Hardware-Grouping groups operation #5 and other nodes, are #2, #3, #6 and #7, as a virtual ISE candidate, i.e. vS_5. Since operation #5 has two hardware implementation options, vS_5 has two evaluation results, namely $vS_{5,1}$ (execution time = 1.7, silicon area = 2400) and $vS_{5,2}$ (execution time = 1.4, silicon area = 3000).

Merit Function

The merit function is divided two parts that are used to calculate software and hardware implementation option, respectively. The merit value ($merit_{x,SW\text{-}i}$) of software implementation option i of operation x is derived with:

$$merit_{x,SW\text{-}i} = merit_{x,SW\text{-}i} \times ET(x,SW\text{-}i) \tag{3}$$

where $ET(x,SW\text{-}i)$ is the time of executing operation x on implementation option (i.e. function unit) i.

In hardware part, the merit function consists of four cases, critical path (case 1), size checking (case 2), constraints violation determination (case 3) and performance as well as area benefits calculating (case 4). Figure 14 shows the merit function algorithm of hardware. As mentioned above, only packing the operation locating on the critical path can have benefit in execution time reduction. Hence, initially, in case 1, the algorithm adjusts the merit value according to the locality of operation. Then, in case 2, the algorithm determines whether $size(vS_x)$, which is the number of operation in vS_x, is equal to 1. Notably, this work assumes that every operation is one-cycle delay in original processor specification. If a multiple-cycle delay is assumed, then case 1 should be tailored to fit this situation. If *size*(vS_x) = 1, then vS_x only has one operation *x* such that the performance cannot be improved. Therefore, the algorithm multiplies the merit value of every hardware implementation option by a constant β_{Size} ($0 < \beta_{Size} < 1$) to lower the chance of it being chosen. The calculation of the merit function is then terminated. If no, then goto case 3.

Case 3 verifies whether vS_x violates input/output port and/or convex constraints. If yes, then the merit value of each hardware implementation option is multiplied by constant β_{IO} and/or β_{Convex} ($0 < \beta_{IO} < 1$ and $0 < \beta_{Convex} < 1$), reducing the opportunity for selecting the hardware implementation option, as in case 2. The calculation of the merit function is then terminated. Since operation *x* may have chance to be grouped in an ISE candidate at the following iterations, the algorithm only divides the merit value of each hardware implementation option by a constant. If the algorithm does not allow the possibility of operation *x* becoming an operation in an ISE candidate, the optimal solution may also be excluded. If no, then enter case 4.

In case 4, the merit value of hardware implementation option *j* ($merit_{x,HW\text{-}j}$, $j > 0$) in operation *x* is computed according to (1) the speedup that can be achieved by $vS_{x,HW\text{-}j}$, and (2) the silicon area utilized by $vS_{x,HW\text{-}j}$. The execution cycle reduction and silicon area of the virtual subgraph $vS_{x,HW\text{-}j}$ is represented by $cycle\_saving_{x,HW\text{-}j}$ and $Area_{x,HW\text{-}j}$, respectively. The main criterions used in case 4 are followings:

(1) If $vS_{x,HW\text{-}j}$ can improve the performance, then all hardware implementation options must have larger merit value than the software one, and the merit value is direct proportion to the execution time reduction.

(2) If vS_x locates on the critical path, the execution time of vS_x should be as short as possible to improve performance.

(3) If vS_x locates on the non-critical path, the execution time of vS_x should be as close to maximal allowable execution cycle (Max_AEC) as possible to save silicon area. The Max_AEC is the difference between the earliest possible execution time of first operation in vS_x and the leatest possible execution time of first operation in vS_x. Restated, there does not have any performance loss, if the execution time of vS_x is equal to or shorter than Max_AEC. Figure 15 is an example. In this example, Max_AEC of ISE (consists of operation 8, 9, 10 and 11) is three cycles.

(4) If two hardware implementation options have same performance improvement, then the one using less silicon area should have larger merit value than another.

Accordingly, in case 4, the algorithm first multiplies the merit value of implementation option j by $cycle\_saving_{x,HW\text{-}j}$. Then, if vS_x locates on critical path, the algorithm continues to determine whether the execution time of implementation option j (i.e. $ET(vS_{x,HW\text{-}j})$) is equal to the maximal execution cycle reduction achieved by vS_x (i.e. $ET(vS_{x,\,HW\text{-}MAX})$). If yes, then the algorithm adjusts the merit value according to the ratio of $Area_{x,HW\text{-}MAX}$ to $Area_{x,HW\text{-}j}$. Here, $Area_{x,HW\text{-}MAX}$ represents the largest silicon area consumed by vS_x. If no, then the merit of implementation option j is divided by the difference between 1+ $ET(vS_{x,HW\text{-}j})$ and $ET(vS_{x,HW\text{-}MAX})$. On the other hand, if vS_x does not locate on critical path, the algorithm uses similar method described as above to compute the merit value.

```
Case 1. (Critical path)
    If (operation $x$ locates on the critical path AND operation $x$ has hardware implementation option)
        $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \div \beta_{CP}$;
Case 2. (The size of $vS_x$ is equal to 1)
    If ($size(vS_x)$ == 1)
    $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \times \beta_{Size}$;
Case 3. (Violate constraints, and the size of $vS_x$ is larger than 1)
    If ($vS_x$ violates in/out constraint)
        $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \times \beta_{IO}$;
    If ($vS_x$ violates convex constraint)
    $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \times \beta_{Convex}$;
Case 4. (Conform with constraints, and the size of $vS_x$ is larger than 1)
    If ($vS_x$ observes in/out and convex constraint AND $size(vS_x) > 1$)
        // Performance improvement check
        $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \times cycle\_saving_{x,HW\text{-}j}$;
        // Hardware usage check
        If ($vS_x$ locates on the critical path)
            If ($ET(vS_{x,HW\text{-}j}) == ET(vS_{x,\,HW\text{-}MAX})$)
            $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \times (Area_{x,HW\text{-}MAX} \div Area_{x,HW\text{-}j})$;
            Else
                $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \div (1 + ET(vS_{x,HW\text{-}j}) - ET(vS_{x,HW\text{-}MAX}))$;
        Else
            If ($ET(vS_{x,HW\text{-}j}) \leqq$ Max_AEC)
            $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \times (Area_{x,HW\text{-}MAX} \div Area_{x,HW\text{-}j})$;
            Else
                $merit_{x,HW\text{-}j} = merit_{x,HW\text{-}j} \div (1 + ET(vS_{x,HW\text{-}j})$ – Max_AEC);
```

Figure 4.3.7: Algorithm of the merit calculation of hardware implementation option

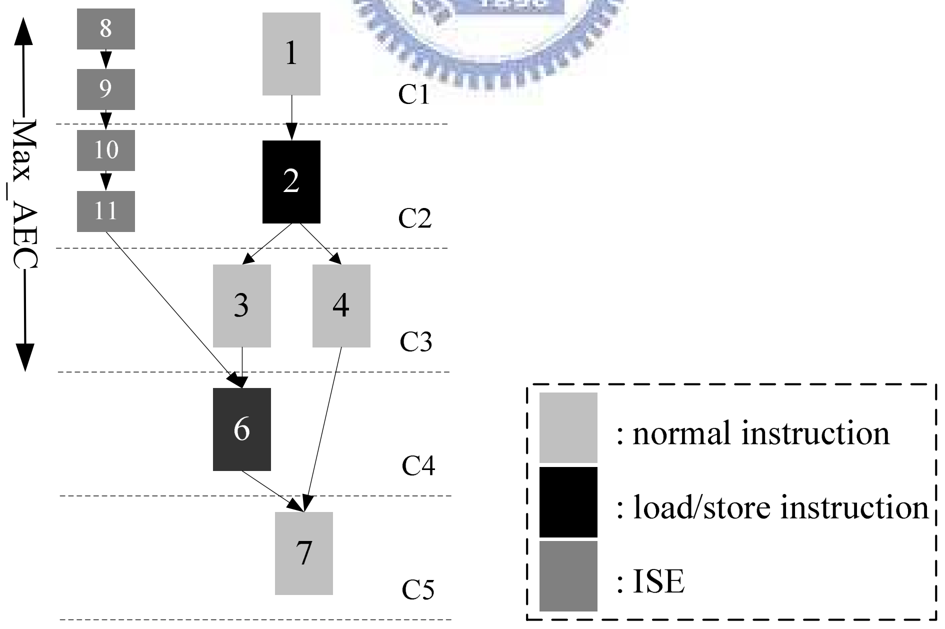


Figure 4.3.8: Example of maximal allowable execution cycle (Max_AEC)

4.4 The Complexity Analysis of ISE Exploration Algorithm

Since the proposed ISE exploration algorithm is terminated until converge, it is very difficult to know how many iterations must be performed before convergence. In this paper, hence, we just analyze the complexity of each step in ISE exploration flow rather than whole algorithm. The complexity of step 1, 7 and 8 (only merit computation) are $O(k(n+m))$, where k ($k > 0$) is number of operation in the DFG and each operation has n ($n > 0$) software implementation option(s) and m ($m > 0$) hardware implementation option(s). Hardware-Grouping (also in step 8) is to check the relation between operations. Each operation in DFG must execute this process and for an operation, up to k operations should be checked. The complexity of Hardware-Grouping therefore is $O(k^2)$. Step 3, 4, 5 and 6 are used to schedule operations, and these steps are derived from the idea of list scheduling. The complexity of this process is therefore same with list scheduling, i.e. $O(k^2)$. Apart from the above steps, the complexity of other steps are $O(k)$. Based on the above analysis, the complexity of executing one iteration is $O(k^2)$.

Chapter 5

Experimental Results

5.1 Experimental setup

The Portable Instruction Set Architecture (PISA) [12], which is a MIPS-like ISA, was employed to evaluate the proposed ISE exploration algorithm and the previous one [8]. Severn benchmarks, including CRC32, FFT, adpcm, bitcount, blowfish jpeg and dijkstra, were used in this simulation. Each benchmark was compiled by gcc 2.7.2.3 for PISA with -O0 and -O3 optimizations. For both ISE exploration algorithms, six cases were evaluated that includes 2-issue with 4/2 and 6/3, 3-issue with 6/3 and 8/4, as well as 4-issue with 8/4 and 10/5. (e.g. 6/3 represents that number of read and write ports of register file are 6 and 3, respectively)

Table 5.1.1: Hardware implementation option setting

Operation	Delay (ns)	Area (μm^2)	Operation	Delay (ns)	Area (μm^2)
add addi addu addiu	4.04	926.33	and andi	1.58	214.31
	2.12	2075.35	or ori	1.85	214.21
sub subu	4.04	926.33	xor	4.17	375.1
	2.14	2049.41	xori	2.01	565.14
mult	5.77	84428	sll sllv srl srlv sra srav	3.00	400.00
multu	5.65	79778.1			
nor	2.00	250.00			
slt slti sltu sltiu	2.64	1144			
	1.01	2636			

In this simulation, we assume that: (1) the CPU core is synthesized in 0.13 μm CMOS technology and executes in 100MHz; (2) the issue width are from 2 to 4; (3) the read/write ports of register file are 4/2, 6/3, 8/4 and 10/5, respectively; and (4) the execution cycle of all instructions in PISA is one cycle, i.e. 10 (ns). Table 1 lists the hardware implementation option settings (delay and area) of instructions in PISA. Significantly, only instructions that can be grouped into ISEs are listed in table 1. These settings were either obtained from [Name author] [14], or modeled by Verilog and synthesized with Synopsys Design Compiler. Additionally, we also assumed both works consider pipestage timing constraint.

Because of the heuristic nature of the ISE exploration algorithm, the exploration was repeated 5 times within each basic block, and the best result among the 5 iterations was chosen. As mentioned before, the results of ISE exploration are only ISE candidates. However, without performing ISE merging and selection, these results cannot be viewed as the final ones. In this paper, therefore, we adopt a greedy method to select ISE(s). After merging ISEs, the ISE selection algorithm ranks ISE candidates according to their performance improvement. By using rank order, ISE selection algorithm then chooses as many ISEs as possible under predefined constraints, such as number of ISEs and silicon area. Finally, we replace the instruction pattern(s) in the program(s) with ISE(s), and schedule the code again to obtain execution time. In this paper, both approaches adopt same ISE design flow, as shown in Fig. 3, and use same ISE selection algorithm.

The parameters adopted in this work and their meanings are listed below.

- α: the relative influence of merit and trail.
- λ: the relative influence of scheduling priority (SP) and merit as well as trail.
- ρ_1, ρ_2, ρ_3, ρ_4 and ρ_5: the evaporating factor in trail update.
- β_{CP} and β_{Size}: the tendency to choose hardware implementation option in a node.
- β_{IO}: the decay speed when the input/output constraint is violated.
- β_{Convex}: the decay speed when the convex constraint is violated.

A large α makes the algorithm converge slowly, while a small α is on the contrary. Restated, a large α obtains a solution slowly, and a small α obtains a poor solution, but quickly. ρ_1, ρ_2, ρ_3, ρ_4 and ρ_5 has same characteristic with α. β_{CP} and β_{Size} determine the chance of an operation, which does not fit in with criterions of ISE selection, being packed into ISE again at following iterations. Similar with β_{CP} and β_{Size}, β_{IO} and β_{Convex} also decide the opportunity of an illegal operation being encapsulating into ISE again at following iterations.

In this experiment, the initial merit value of the software and hardware implementation option was 100 and 200, respectively; the initial trail value of all implementation options were 0; P_END was 99%. The probability value adopted α = 0.25, the evaporating factor ρ_1, ρ_2, ρ_3, ρ_4 and ρ_5 are 4, 2, 2, 2 and 0.4, respectively, and the merit function had β_{CP} = 0.9, and β_{Size} = 0.7, β_{IO} = 0.8 and β_{Convex} = 0.4.

5.2 Experimental results

Figures 16 and 17 depict the average execution time reduction under silicon area and number of ISEs constraints, respectively. Each bar in Fig. 16 comprises several segments, which indicate different silicon area constraints, are 20000, 40000, 80000, 160000 and 320000 μm^2; while, the segments in Fig. 17 means number of ISEs, are 1, 2, 4, 8, 16 and 32 ISEs. The first word of each label on X axis in both Figs. 16 and 17 indicates which ISE exploration algorithm is adopted. “MI” and “SI” denote the proposed ISE exploration algorithm and that of Wu [8], respectively. The symbols in parentheses of each label on the X-axis are the number of register file read/write ports in use, issue width, and which optimization method (-O0 or -O3) is used. For instance, (4/2, 2IS, O3) means that the register file has four read ports as well as two write ports, issue width is two and that the O3 optimization method is employed.

Obviously, under same silicon area constraint, our proposed algorithm exhibits better execution time reduction than [8] in all cases. In Fig. 16, for both algorithms, O3 exhibits better execution time reduction than O0 in cases of 2IS. This is because O3 often uses various compiler optimization techniques. Some of these techniques (like loop unrolling, function inlining, etc.) remove branch instructions, and increase the size of basic blocks. The bigger basic block usually has a larger search space, such that it has a greater opportunity to obtain the ISEs, which have more execution time reduction. However, O0 exhibits better execution time reduction than O3 in cases of 3IS. Possibly, because O3 increases instruction-level parallelism, most instructions are executed on ALUs such that less performance improvement is achieved in O3. In 4IS, since the issue width is large enough, instruction-level parallelism can be easily attained, even without any compiler optimization techniques. The performance gap between O0 and O3 is therefore not obvious. Fig. 17 depicts the execution time reduction for different number of ISEs. Same with Fig. 16, Fig. 17 also has similar results. In all cases, our proposed algorithm significantly outperforms than [8].

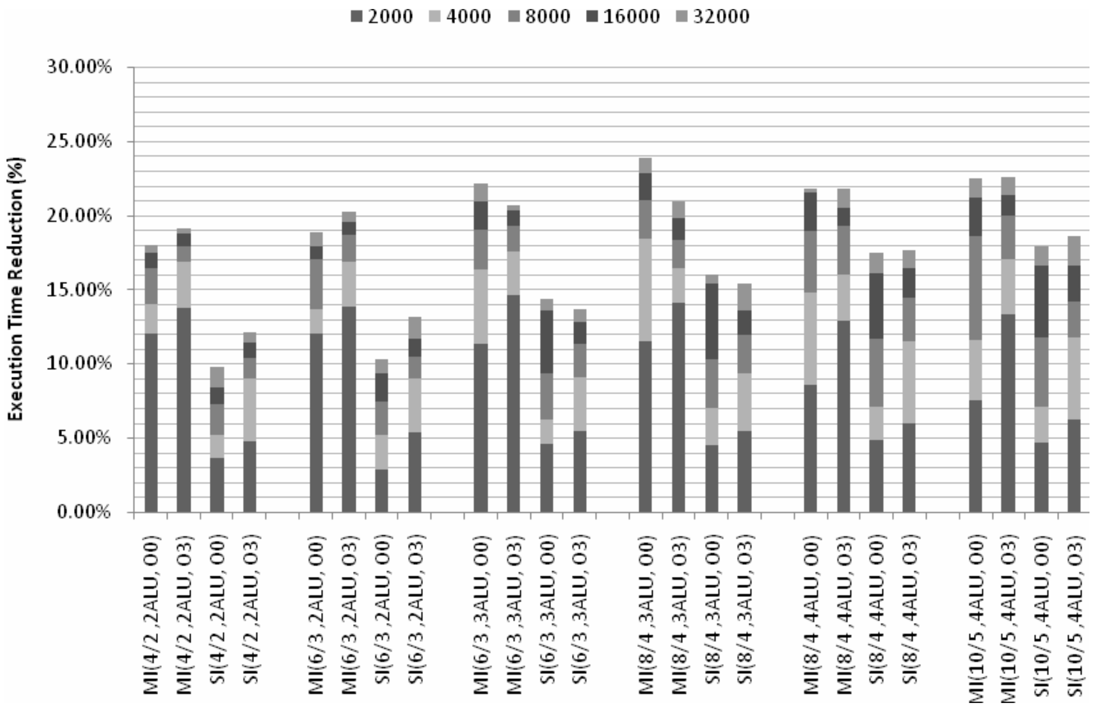


Figure 5.2.1: Execution time reduction under different silicon area constraints

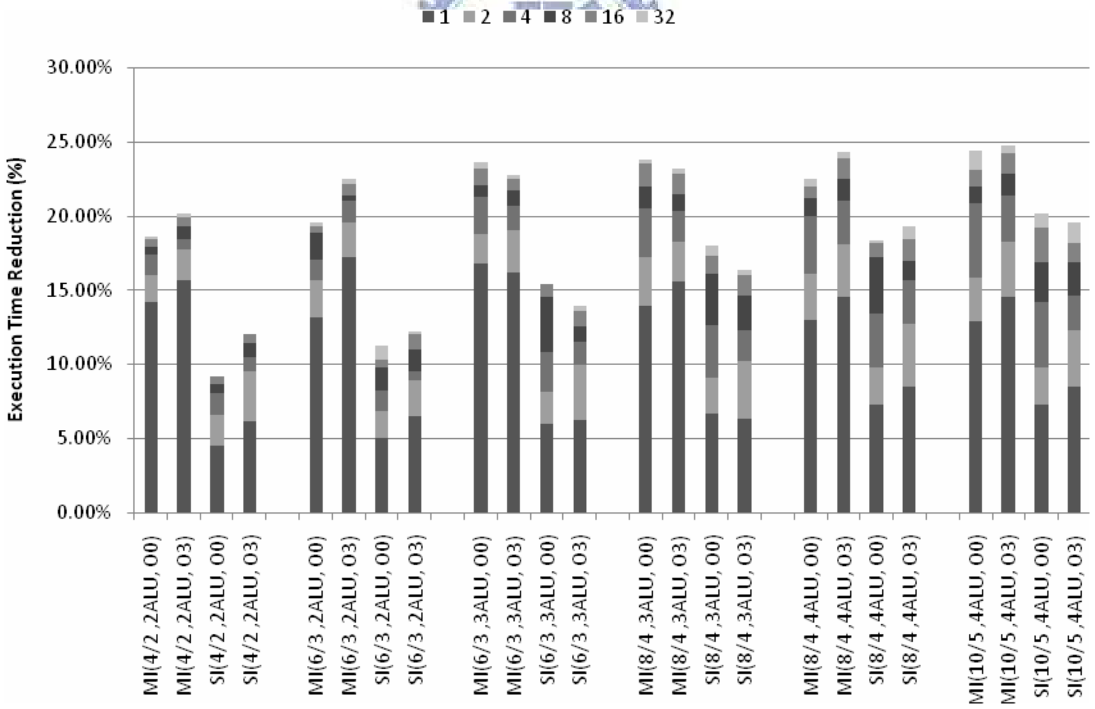


Figure 5.2.2: Execution time reduction for different number of ISEs

The comparison between the proposed algorithm and [8] in silicon area cost and execution time reduction is illustrated in figure 18. In Fig. 18, we clearly observe that most of execution time reduction is dominated by several ISEs, especially first ISE. In other words, the number of ISE is not entirely proportional to the execution time reduction. This is because for most programs, their execution time is usually concentrated in small number of basic blocks, i.e. hot basic blocks. Hence, although increasing the number of ISEs can boost performance, but considerable silicon area cost must be incurred.

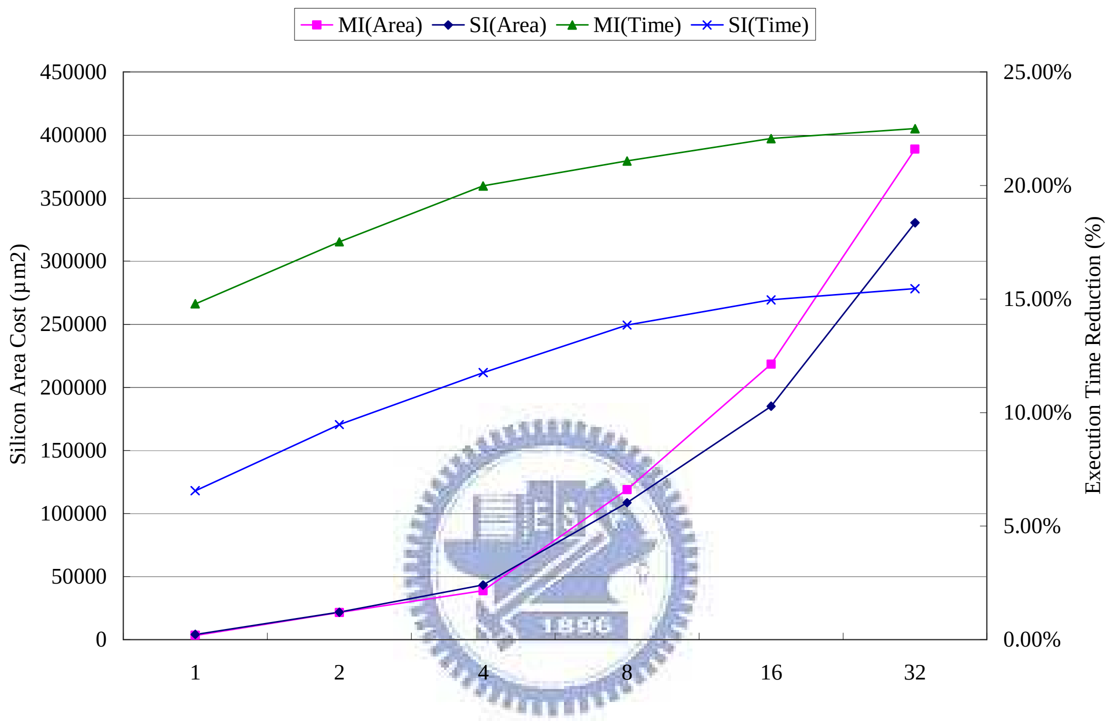


Figure 5.2.3: Silicon area cost v.s. execution time reduction

Chapter 6

Conclusion

The proposed ISE generation algorithm can significantly reduce execution time for the multiple-issue processor. Previous studies in ISE exploration only take the legality of operation into account. However, in multiple-issue processor, only considering the legality of operation cannot gain much execution time reduction and may waste silicon area. To avoid such situation, this work considers not only the legality of but also the locality of operations. Experiment results demonstrate that when only one ISE is used, the proposed design can reduce execution time by up to 17.17%, 12.9% and 14.79% (max., min. and avg., respectively) as compared with the multiple-issue processor without using ISE. Furthermore, under same area constraint our approach has 11.39%, 2.87% and 7.16% (max., min. and avg.) of further reduction in execution time over the previous one[8].

Additionally, we recommend addressing several issues in future work. First, because we put emphasis on exploring ISE rather than on determining the scheduling priority, this paper adopts only simple way (i.e. number of child operations) to determine the scheduling priority. However, many studies shown that different scheduling priority functions would result in different results, i.e. different critical path. Adopting different priority functions to identify the critical path would be interesting to study. Second, the problem [16 and 17] consisting of hardware-software partitioning, hardware design space exploration and scheduling is similar with our work. (hardware-software partitioning $\leftrightarrow$ determining hardware or software implementation options, hardware design space exploration $\leftrightarrow$ selecting an implementation option, and scheduling $\leftrightarrow$ identifying the critical path) Hence, by a slight modification, the proposed ISE exploration algorithm can be adopted to this problem.

Reference

[1] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant System: Optimization by a Colony of Cooperating Agents", *IEEE Transactions on Systems, Man and Cybernetics*, February 1996.

[2] ARM Cortex-A8, http://www.arm.com/products/CPUs/ARM_Cortex-A8.html

[3] CEVA: CEVA-X1620 Datasheet. CEVA, 2004

[4] Laura Pozzi, Kubilay Atasu, and Paolo Ienne, "Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets", *IEEE Tran. on CAD*, July, 2006.

[5] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Laura Pozzi, and Paolo Ienne, "Fast automated generation of high-quality instruction set extensions for processor customization", *3rd Workshop on Application Specific Processors*, September, 2004.

[6] N. T. Clark, H. Zhong and S. A. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration", *IEEE Tran. on Computers*, October, 2005.

[7] K. Atasu, G. Dundar, and C.Ozturan, "An integer linear programming approach for identifying instruction-set extensions", *CODES+ISSS*, September, 2005.

[8] I-Wei Wu, Shi-Jia Huang, Chung-Ping Chung, and Jyh-Jiun Shann, "Instruction set extension generation with considering physical constraints", *HiPEAC*, January, 2007.

[9] Pan Yu and Tulika Mitra, "Characterizing embedded applications for instruction-set extensible processors", *DAC*, June, 2004.

[10] Pan Yu and Tulika Mitra, "Satisfying real-time constraints with custom instructions", *CODES+ISSS, September*, 2005.

[11] Jason Cong, Yiping Fan, Guoling Han and Zhiru Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures", *20th FPGA*, 2004.

[12] Samik Das, P. P. Chakrabarti, Pallab Dasgupta, "Instruction-Set-Extension Exploration Using Decomposable Heuristic Search", *VLSI Design*, 2006.

[13] Huynh Phung Huynh, "A Survey of Custom Instruction Identification and Selection Techniques".

[14] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling", *IEEE Computer*, 2002.

[15] A Lindstrom and M. Nordseth. Arithmetic Database. Available: http://www.ce.chalmers.se/arithdb/

[16] K. Chatha and R. Vemuri, "An Iterative Algorithm for Hardware-Software Partitioning, Hardware Design Space Exploration and Scheduling", *Design Automation for Embedded Systems*, vol. 5, pp. 281--193, 2000.

[17] Kalavade A, Lee EA. “The extended partitioning problem: hardware/software mapping, scheduling, and implementation-bin selection”. *Design Automation of Embedded Systems*, 1997,2(1):125-163.