國 立 交 通 大 學

資訊工程學系

碩　士　論　文

由執行資料引導的草稿記憶體動態載入程式之研究

Profile-directed Dynamically Loading of Program into Scratch-pad Memory

研究生：宋宜叡

指導教授：鍾崇斌 教授

中 華 民 國 九 十 三 年 六 月

由執行資料引導的草稿記憶體動態載入程式之研究

Profile-directed Dynamically Loading of Program into Scratch-pad Memory

研 究 生 ： 宋 宜 叡 Student: I-Jui Sung

指導教授：鍾崇斌 教授 Adviser: Dr. Chung-Ping Chung

國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science and Information Engineering

July 10, 2004

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 三 年 六 月

To Adele

由執行資料引導的草稿記憶體動態載入程式之研究

學生：宋宜叡　　　　　　　　　　　　　指導教授：鍾崇斌 博士

國立交通大學資訊工程學系碩士班

摘　　要

有效率地利用晶片上的記憶體一直是改進程式執行效能的重要方法。關於快取記憶體(Cache)的研究已經進行數十年，然而草稿記憶體 (Scratch-pad memory)由於其較快取記憶體低的功耗與成本，近年亦吸引許多研究者投入。由於草稿記憶體的特性使然，為使一般的程式 達到有效利用該記憶體，需要從軟體層次進行修改。因此，為了使已撰寫好的程式能夠利用草稿記憶體，許多編譯器的研究者提出從編譯器層次 修改目的程式以利用草稿記憶體的方法。

在本論文中，我們提出一軟體協助之動態載入程式片段進入草稿記憶體以提升程式執行效能的方法。前人提出的動態方法雖可找出最適合動態載入的片段，但由於使用整數線性規劃的緣故，該方法並不適合較大規模的程式。因此在本論文中，我們提出並評估一個基於「分而治之」概念的「試探法」(Heuristic)，用以找出適於動態載入的程式片段。而本試探法中的「分」即將巢狀迴圈(Nested Loops)分解；而「治」的方法則是將問題建模(Model)為一維零/壹背包問題(One-dimensional 0/1 Knapsack Packing Problem)。

實驗結果顯示我們的試探法的表現超越前人所提出之靜態載入方法，並且其性能可與傳統I-Cache抗衡。因此我們認為本論文提出的乃是一個適合較大型程式動態、有效率地利用草稿記憶體的可行方法。

Profile-directed Dynamically Loading of Program into Scratch-pad Memory

Student: I-Jui Sung Adviser: Dr. Chung-Ping Chung

Department of Computer Science and Information Engineering
National Chiao Tung University

ABSTRACT

Effective utilization of on-chip memories has always been an important factor to improve the performance of a program. Caching techniques has been studied for decades, whilst recently scratch-pad memory is getting more and more attention, because of its lower cost and lower power-consumption compared to the cache.

Due to the nature of scratch-pad memory, it is required to modify current program to effeciently utilize the scratch-pad memory. Hence, for legacy programs, compiler researchers have proposed automatic loading methods of source program into scratch-pad memory. However, the ILP framework employed by previous study prevents efficient application of such method for larger programs.

In this thesis, we focus on the dynamic loading of parts of program to on-chip scratch-pad memory to improve the performance.

Previous dynamic approach applies the ILP framework to solve the problem of deciding optimal set of dynamically-loaded program parts, but it is not feasible when program size grows. Hence, a heuristic to solve the problem of deciding dynamically loading parts of a program into scratch-pad memory is proposed and evaluated in this thesis. Our heuristic is based on the idea of ‘divide-and-conquer’. Here ‘divide’ means the decomposition of nested loops, and ‘conquer’ means to an one-dimensional 0/1 knapsack-packing method.

Evaluation results shows that our heuristic outperforms the static approach, and the performance of software-controlled dynamically-loaded code in scratch-pad memory produced by our method is comparable to traditional I-cache. This suggests a viable path for larger applications to efficiently utilize the scratch-pad memory.

致　　謝

在此我要感謝我的指導老師：鍾崇斌老師。沒有老師嚴謹的指導，我不可能完成這篇論文。老師給我的不僅僅是學業上的指導，亦包含了許多與人相處的智慧。

另外，我想感謝本實驗室的另一位大家長—單智君老師。單老師除了擔任我的口試委員以外，在我的兩年碩士生活之中，也給了我許多寶貴的指導。還有，擔任另一位口試委員的李政崑老師。李老師在口試我以後亦不吝給予指點，讓我的碩士論文能夠更加的完整。

還要謝謝實驗室的博士班學長們，特別是鄭哲聖學長。學長不但不厭其煩的指出我論文不足之處，在我每次打斷學長工作發問時，也隨時都和氣地與我討論。當然，實驗室所有的同學、學弟們與我一起渡過的時光以及一起討論課業的情景，我永遠不會忘記。

最後我要感謝父母及我的女友：沒有你們無條件的溫柔支持，憑我是不足以完成這份論文的。

謝謝你們！

宜叡

2004年6月30日

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The idea of using a small, fast memory to store frequently referenced code or data is not new. Caching techniques has been studied for decades, whilst recently scratch-pad memory is getting more and more focuses on it, because of its lower cost and lower power-consumption comparing to the cache.

The scratch-pad memory has a two-dimensional array structure, often consists of SRAM memory cells. Like conventional memories, it access the memory cells through the decoding logic and the column circuitry. Usually the scratch-pad memory is located on-chip, as in Figure 1.1.

In this thesis, we assume that the scratch-pad memory occupies one distinct part of the entire memory addressing space, while access to the rest part of the address space will go via the external bus. In short, the scratch-pad memory in our model can be viewed as a fast on-chip memory with the same way to access like main memory, but the nature of on-chip SRAM makes its access faster than external memory (especially when DRAM is employed as main memory). Therefore, any use of scratch-pad memory shall be accomplished explicitly via software. Also, due to the cost of real estate on the silicon, the capacity of typical scratch-pad memories fall in the range of several kilobytes.

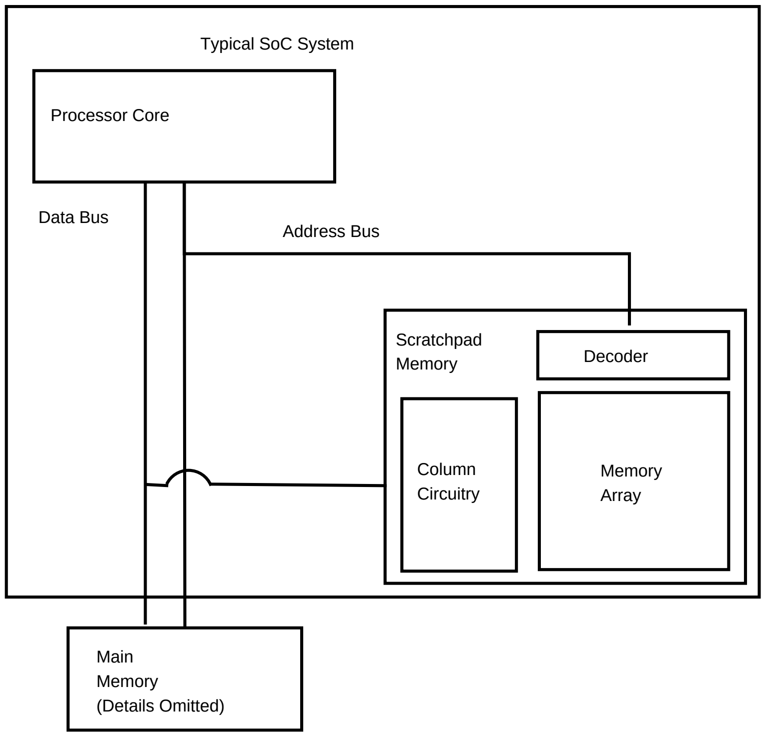

Figure 1.1: Block diagram for Scratch-pad Memory and its relative location to processor core and external memory

A comparison between scratch-pad memory and cache memory is briefed in Table 1.1. Further qualitative and quantitative comparisons between scratch-pad memory and cache memory are available in Banakar et al. [2001].

In general, it is the responsibility for a compiler to select parts of program or data to be placed in the limited on-chip scratch-pad memory. In this paper we mainly focus on a static, heuristic selection of program parts to be loaded into scratch-pad memory dynamically during execution.

Table 1.1: Comparison between Cache and Scratch-pad Memory

	Cache	Scratch-pad Memory
Accessing	Image of Main Memory	Disjoint Address Range from Main Memory
Area Cost [See Banakar et al., 2001]	1	0.66
Loading Time	Dynamic	Static or Dynamic
Loaded by	Hardware	Software
Transparency	Transparent to Software	Requires Explicit Modification in Software

1.1 Related Work

Previous works [Banakar et al., 2001, Steinke et al., 2001, 2002] on the utilization of scratch-pad memory focus on the energy optimizations. Though their objective is mainly on reducing energy usage, the reduction of program execution time is also mentioned in Steinke et al. [2001, 2002]. In [Hajj et al., 2000], the problem of packing loops into a dedicated simple cache (called L-Cache) is studied. Though the underlying hardware is different from the scratch-pad memory, their formulated problem is quite similar to our problem in a higher abstraction level, and their solving techniques provides a different approach to this probelm. Basically their heuristic approach is to greedily select these basic blocks to be placed into the L-Cache, based on their execution frequencies and (nested) loop depths.

In this paper, there are two approaches directly related to our work: a static and a dynamic approach. They are described in section 1.1.1 and section 1.1.2.

1.1.1 Static Approach

In Steinke et al. [2002], a static packing of program parts and variables into scratch-pad memory are presented. The selected program parts are never changed during the lifetime of program execution in this study. As there might be several hot-spots in program, Steinke et al. [2001] mentioned that such static approach may suffer from the constraint which all hot-spots must be packed into the scratch-pad memory at the same time.

The packing algorithm used in Steinke et al. [2002] is based on the one-dimensional 0/1 knapsack packing, where the knapsack capacity is set to the size of scratch-pad memory. Items to be packed into scratch-pad memory are the variables and basic blocks in the program. Then, a value is assigned to each item based on its number of access (for the variable case) or execution (for the case of program code) times the size of this item. Finally a knapsack packing solving algorithm is applied to determine the best static assignment of program codes and variables in scratch-pad memory.

1.1.2 Dynamic Approach

A 'dynamic' packing algorithm is proposed in Steinke et al. [2001]. The word 'dynamic' means the selected program parts in scratch-pad memory may be dynamically-loaded, although the definition of these selected program parts is still accomplished at compile time. In that study, they extend the work in Steinke et al. [2002], letting each selected program parts pairing a 'copy function' to load the selected program parts at the point of copy function into scratch-pad memory. The principles of selecting copy function for a selected program part in Steinke et al. [2001] can be summarized as follows:

1. A selected program part can be assigned exactly one of the copy functions located at its surrounding loop entries.
2. The program parts that share the same copy function are copied to the scratch-pad memory simultaneously.
3. Different copy functions may share the scratch-pad memory at the same time, but the sum of all coexisting copies of selected program parts can not exceed the capacity of scratch-pad memory.

These principles are then formulated as integer linear programming (ILP) [Nehmhauser and Wolsey, 1988] constraints. Then, the optimal selection of program parts, along with the locations of their copy function, and allocation of scratch-pad memory is decided using an ILP solver simulataneously.

Though significant improvements are observed using this approach for small programs, the NP-completeness of solving integer linear programming problems [Nehmhauser and Wolsey, 1988] makes the application of their methodology feasible only to small problem size.

1.1.3 Summary on Related Work

In the view of problem solving, the static approach [Steinke et al., 2002] models the static packing problem as an one-dimesional 0/1 knapsack packing problem (i.e., only single knapsack), while the dynamic approach in Steinke et al. [2001] models the dynamic packing as a multidimensional knapsack packing problem. The one-dimensional version mentioned above has a well-known polynomial time solution using dynamic programming, while the multidimensional knapsack packing problem is currently solvable

Table 1.2: Comparison between Static, Dynamic [Steinke et al., 2001], and Our Approach

	Static Approach	Dynamic Approach
Problem Modeling	One-Dimensional 0/1 Knapsack Packing	Multi-Dimensional 0/1 Knapsack Packing
Subject	Basic Blocks	Basic Blocks, packed in different loops
Solution	1-D 0/1 Knapsack Packing	ILP Solver
Solution Quality	Optimal	Optimal

through ILP framework. A comparison between these methods, as long as the approach presented in this thesis is listed in Table 1.2.

1.2 Motivation and Objectives

As mentioned above, dynamically approach [Steinke et al., 2001] models the problem as a multidimensional packing problem, solvable by an Integer Linear Programming (ILP) Solver, whilst the number of knapsacks and capacity of each knapsack is also required to be solved at once. The solution of the multi-dimensional knapsack packing problem whill help determining the optimal set of selected program parts and their corresponding copy function. However, this approach is NP-complete, which is not feasible when program size grows.

To deal with this issue, this thesis aims to construct a divide-and-conquer heuristic approach for this problem, based on a few observations which are described later in chapter 2.

Unlike previous works which focus on energy savings, this thesis aims at reducing the total execution time of program by utilizing scratch-pad memory through software means to copy program code into it.

1.3 Organization of This Thesis

This thesis is organized as follows: In chapter 2 we put emphasis on the approach employed in this paper. Chapter 3 evaluates our design, with compare to the static approach and cache. Finally, chapter 4 concludes this thesis.

Chapter 2

Approach

In this chapter, we discuss the approach employed in this thesis and the design of our divide-and-conquer heuristic to the dynamic loading problem. This chapter is organized as follows: section 2.1 presents a few observations to the nature of dynamic-loading problem and how to divide this into subproblems, section 2.2 then propose the solution for these subproblems. Section 2.3, section 2.4, and section 2.5 discusses the motivation and method to further partition these subproblems. In section 2.6 an algorithm is proposed to solve the problem using decomposition method and knapsack packing. Finally, section 2.7 proposes a mechanism achieving dynamically loading, which is also used in our evaluation in next chapter.

2.1 Some observations to the Dynamically-Loaded Code

In order not to confuse reader with the term of 'selected program part' mentioned above, we shall define a term here to describe the program parts selected for our approach to dynamic loading into scratch-pad memory. In the following text, we shall call it using the term of 'dynamically loaded code'. Informally, a dynamically loaded code is a piece of program which shall be loaded into scratch-pad via software assist before its execution. Dynamic loaded code need not to be continuous in the control flow graph of program it belongs to. Instead, they are merely a collection of some basic blocks, but are loaded into scratch as a whole during the transfer of control from somewhere outside the dynamically loaded code.

With the following observations, the goal of this thesis is to construct a divide-and-conquer heuristic approach to this problem. Here we have the following observations to the properties of dynamically loaded code:

1. A dynamically loaded code is worthy of loading only if it is inside one or more loops.
2. The set of suitable reloading points of a dynamically loaded code should be the entries of its surrounding loop(s).
3. For sufficiently large scratch-pad memories, the best reloading point of a dynamically loaded code is at the entry point of the outermost loop it belongs to.

The first observation is due to the software controlled nature of scratch-pad memory. Unlike cache, the loading of code into scratch-pad memory entirely relies on another piece of software. Hence, a higher loading cost is expected. It follows that only a piece of code residing in a loop can offer sufficient temporal and spatial locality to take advantage of being software-controlled dynamically-loaded into scratch-pad memory.

Before explaining observation 2, we shall first define the *reloading point* of a given piece of code. The reloading point of a piece of code is a point in the program which does the job of loading the code into scratch-pad memory. Since a reloading creates major overhead, the number of re-loadings must be at small as possible.

Hence the reloading point of a dynamically loaded code should at least be the entry of the closest surrounding loop, and in most cases the outer loop entry the reloading point resides at, the smaller total reloading cost will be. In general the outer loop executes less frequently than the inner loop, and the entry of outermost loop of a dynamically loaded code is only executed once. If the reloading point of a dynamically loaded code has been placed at the outermost surrounding loop, it will require minimal reloads. Also, a reloading point prior the entry of the outermost loop of its corresponding dynamically loaded code virtually creates no extra benefits but occupation of precious scratch-pad memory. Thus we have the observation 3.

The three observations above can give us a rough image of the way to divide the problem: Since all basic blocks inside a nested loop do not have to put their reloading points prior to the entry of outermost loop they belongs to, two disjoint nested loops need not to share the same reloading point. Hence it is then reasonable to divide the problem of deciding set of dynamically-loaded code and their corresponding reloading points to subproblems consists of outermost loops in the program. Also, since the set of of each subproblem (i.e., disjoint nested loop) does not overlap, each subproblem can then be independently solved. After each subproblem is solved (i.e., the best set of basic blocks and loading points are decided), the problem is solved.

It is worth noting that, as observation 3 suggests, each solution of a subproblem is mapped to at least one dynamically-loaded code. In next section we shall describe the way to 'conquer' the subproblem we defined above. In section 2.3, we shall follow these observations and further discuss them.

2.2 Solving subproblems

In previous section we have discussed a way to partition the problem (the basic partitioning), but the solving method for each subproblem is yet discussed. In this section, we shall present the way to solve such problem.

The main issue of Steinke et al. [2001] is that they modeled the problem into a multi-dimensional knapsack packing problem, which is NP-complete.Hence we shall try another approach:

- Define each subproblem as a unit of dynamically-loaded code.
- Pack each subproblem using one-dimensional, 0/1 knapsack packing independently.
- If a subproblem shows it may be beneficial to have more than one parts of dynamically-loaded code, further divide the subproblem

This approach keeps the packing algorithm considering only one packing problem at a time. Hence the knapsack size shall be the entire scratch-pad memory. And the problem deciding the number of knapsacks are reduced to the problem of deciding the partitions. Note that we do not constrain ourself to the basic partitioning mentioned above. Instead we consider the possibilities to further divide a partition, creating better chance for dynamically-loading of code into scratch-pad memory.

We shall describe the way to further divide a subproblem (partition) in section 2.4 and section 2.5

2.3 How to Determine a Partition of Problem?

We define a partition of problem as a subproblem discussed in the last paragraphs of section 2.1. As section 2.1 suggests, such a partition are the outermost loop in program. We can define 'the outermost loop' more formally as follows:

The set of nested loops of a program are the subset of the strongly-connected component [Tarjan, 1972] of the control flow graph of the program.

Definition: A strongly connected component is a maximal subgraph of a directed graph such that for every pair of vertexes u, v in the subgraph, there is a directed path from u to v and a directed path from v to u.

The set of nested loops are the subset of the set of strongly-connected components of the control flow graph, because any statements which do not belong to any loop are considered itself as a strongly-connected component. However such statement is not worthy for dynamically loading since no locality presents for these statements. For convenience, we shall ignore these 'orphan' statements in later discussions.

Now, we can say a partition defined above is a strongly-connected component of control flow graph. We define such partitioning method as the *Basic Partitioning* of problem. According to observation 3 in section 2.1, the basic partitioning shall be the best partitioning if scratch-pad memory have sufficient capacity to load the maximum-sized partition.

2.4 Discussion on SCC

In previous section, we have defined the basic partitioning of the problem as the strongly-connected component of control flow graph of the program. In this section, we discuss some issues caused by scratch-pad memory capacity constraints. Consider a nested loop with two inner loops, which behave as "Hot-Spots [1]," as shown in Figure 2.4.

Assume that our scratch-pad memory do not have enough capacity to hold both hot-

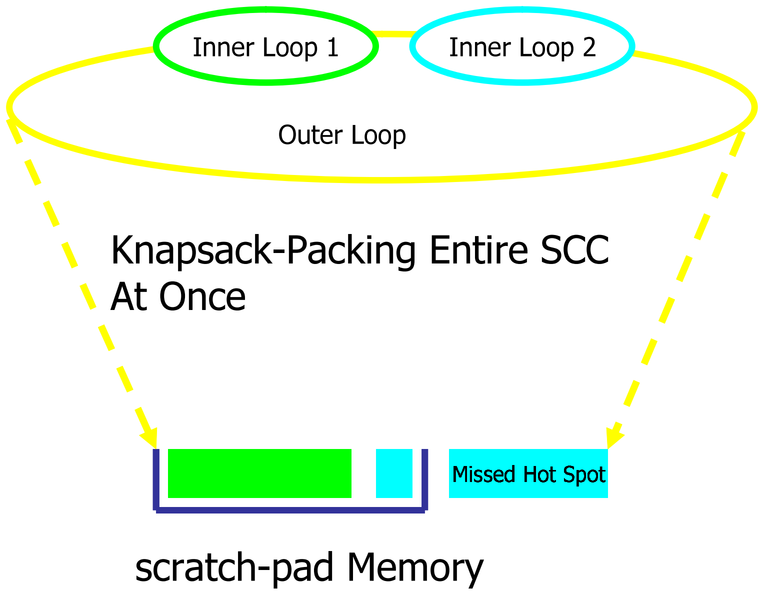

Figure 2.1: Nested Loop, with two frequently executed inner loops

spots. Since the knapsack- packing is applied on the whole partition (i.e., on the set of basic blocks belongs to the partition), there must be some part of the hot-spots lost the chance to be selected into the scratch-pad memory. However, suppose the outer loop is omitted as in Figure 2.2, there do not exist a path from "Inner Loop 2" to "Inner Loop 1", hence "Inner Loop 2" and "Inner Loop 1" falls in different strongly-connected components of the control flow graph. This means they falls in different partition of problem, hence they are then knapsack-packed independently. As a result, at run time,

[1]Parts of code with high temporal locality. Often in the form of tight loop.

the dynamically-loaded “Inner Loop 1” and “Inner Loop 2” are using the scratch-pad memory in turns. If the loop iteration count for both inner loops are large enough, the benefit from execution in scratch-pad memory could balance the cost of reloading both loops.

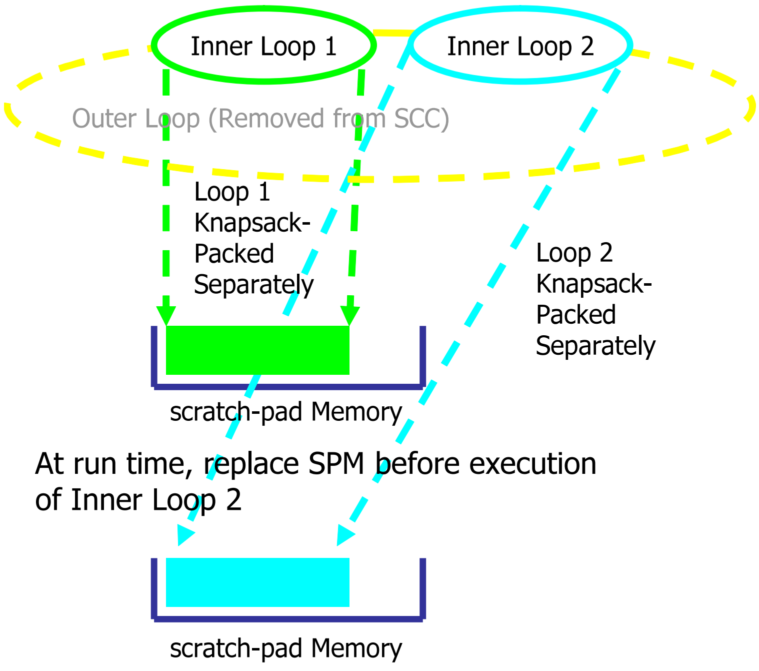
Inner Loop 1
Inner Loop 2
Outer Loop (Removed from SCC)
Loop 1 Knapsack-Packed Separately
Loop 2 Knapsack-Packed Separately
scratch-pad Memory
At run time, replace SPM before execution of Inner Loop 2
scratch-pad Memory

Figure 2.2: Exposed Inner Loops

2.5 Further Partitioning

In section 2.4, an example has been shown to illustrate the potential benefit by remove outer loop in a large strongly-connected component. Then, two subsequent questions arise:

1. When to break the outer loop? (i.e., when to further partition a partition?)
2. How to break the outer loop?

We shall answer these two questions in section 2.5.1 and section 2.5.2.

2.5.1 Decompose Nested Loops

In this section, we discuss the means to break the "outer loop" of a program partition. Effectively, it shall decompose the corresponding strongly-connected component of the partition. That is, we further partition a partition of packing problem by decomposing it. First we have the following assumption:

In the case of nested loops, outer loop is executed less frequently than inner loops.

This assumption holds in most cases, except for the case when inner loop is guarded by a rarely-hold conditional (e.g. guarded by a seldom hold *if() ...* construct in the C programming language). Then we can break the outer loop by finding and removing the arc with fewest execution count in the control flow graph of the strongly-connected component corresponding to the problem partition of interest. Figure 2.3 illustrates this.

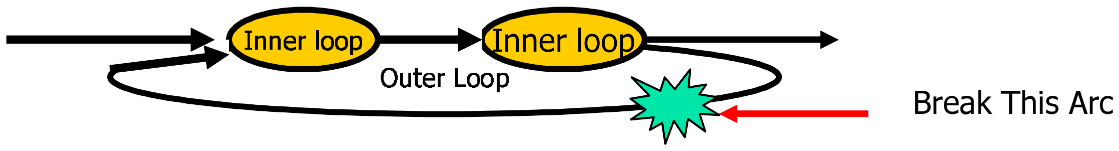

Figure 2.3: Decompose Nested Loops by Breaking Outer Loop

After breaking outer loop, we can then re-apply the algorithm finding all strongly-connected component in the modified control flow graph. In most cases this will cause the inner loops to be exposed, if they present. If there is no inner loops beneath the outer loop, decompose such loop will cause it to become a series of statements. The latter case, however, can be avoided if we can estimate the potential benefit from decomposition, and avoid any blind decomposition if there exists no benefit anyhow. The way to estimate the effect after decomposition will be discussed in section 2.5.2.

2.5.2 Observations on the Decomposition Method

As mentioned above, our "decomposition" is to decompose the strongly connected components of the control flow graph, in a progressive and selective fashion. Recall that a strongly connected component of the control flow graph is our basic partition of the problem. Then, the following observations are made:

1. Further partitioning can reduce the size of a partition, with the cost of extra re-loadings.
2. When a partition already fits in the scratch-pad memory, no further partitioning is necessary.

Observation 1 holds when a decomposition breaks original nested loops into several non-overlapping loops or statements. Or, in the terms of graph theory, when a decomposition effectively breaks original SCC in the control flow graph into several SCCs in the modified control flow graph. Since each SCC in modified control flow graph now possess more reloading chances, extra loadings can be expected.

It is worth noting that sometimes a single step of decomposition may not really decompose a strongly-connected component if there still exists a path of execution between each pair of nodes inside the strongly-connected component after decomposition.

Based on observation 1, we can see that if a partition already fits in the scratch-pad memory, the reloading count of every basic blocks inside the partition is at most the number of entries of the SCC representing the partition. (Branch with its branch target SCC equal to the SCC in which this branch site resides does not cause a reloading)

However, if such partition is further partitioned, the reloading count of each basic block can increase whilst no more basic blocks get the benefit from being dynamically loaded into scratch-pad memory since every basic block can already be put into scratch-pad memory simultaneously before further partitioning. This argument is the basis of observation 2.

In other words, further partitioning a partition can be beneficial if the following conditions are met:

1. It increases the potential number of basic blocks to be dynamically loaded into scratch-pad memory.
2. The reloading incurred by further partitioning does not exceeds the benefit from 1.

We shall describe the metric used in condition 2 in section 2.5.3. Note that if the partition is knapsack-packed, then it must fit in the scratch-pad memory, hence observation 2 also holds for this case. That is, a knapsack-packed partition is unnecessary to apply any further partitioning on it (i.e., more decomposition of the SCC corresponding to this partition in the (possibly modified) control flow graph is not necessary).

2.5.3 The Metric

To decide on which step to apply on a partition (either decomposition of the corresponding SCC of the partition or knapsack-packing its basic blocks), a unified metric is required to estimate the net benefit earned by either decomposition or knapsack packing. In this section, the metric used in our algorithm is presented.

The idea is to define estimated total earned cycles if we dynamically load a partition into scratch-pad memory. Since decomposition breaks original partition into smaller partitions whilst knapsack packing may "strip" the original partition, we shall simply evaluate the effect of decomposition by summing up the net benefit of all partitions created by decomposing current partition.

So we can then define the net benefit of a partition p as follows:

$$\textit{NetBenefit(p)} = \textit{Profit(p)} - \textit{Cost(p)}$$

To make things clear, first we define the *Memory Speed Gap* to be the slow down of main memory from scratch-pad memory, minus 1:

$$\textit{Memory Speed Gap} = \frac{\textit{Access Latency of Main Memory}}{\textit{Access Latency of Scratch-pad Memory}} - 1,$$

and $TotalExecutedInstructions(p)$ being total executed instructions in a partition p:

$$\textit{TotalExecutedInstructions(p)} = \Sigma\{\textit{Size of B}\times\textit{Execution Count of B} \mid \textit{Basic Block } B \in p\}$$

then the profit obtained from loading a partition p into scratch-pad memory can be defined as follows:

$$\textit{Profit}(p) = (\textit{Memory Speed Gap}) \times (\textit{TotalExecutedInstructions(p)}),$$

also the cost:

$$\textit{Cost}(p) = (\textit{Number of Reloading of partition } p) \times (\textit{Reloading Cost of partition } p)$$

It is worth noting that, since decomposition method introduce more partitions by dividing the current partition, the estimated benefit of decomposition of such partition shall be the sum of *NetBenefit(p)* of each partition p created from decomposition, with non-negative net benefit. The reason why only these partition with non-negative net benefit is counted is because only those partitions with non-negative net benefit are capable to be loaded dynamically in scratch-pad memory. That is, only these partitions with non-negative net benefit for dynamic loading will be chosen by the compiler as the potential dynamic loading objects. These "dynamic loading objects", and the corresponding dynamic loading mechanism are described in section 2.7.

Finally, with the function evaluating the performance of a given partition of problem, we are prepared to introduce our algorithm in section 2.6.

2.6 The Algorithm

In this section we introduce the algorithm employed to solve the problem in a divide-and-conquer fashion. As mentioned above, the 'divide' is accomplished by the process of dividing problem into subproblems (i.e., the process of decomposing an SCC), and further divide these subproblem (i.e., further decompose the SCC in modified control flow graph) if further benefit can be expected by dynamically-loading parts of the subproblem in turn. Then, each partition of problem is solved by one-dimensional 0/1 knapsack packing algorithm.

As stated in section 2.5, we shall begin further dividing of subproblems when it has more benefit than just knapsack-pack it. And the metric used to evaluate both steps (decomposition or knapsack-packing) are also introduced in section 2.5.3.

Now we are ready for the algorithm:

Algorithm:

Iteratively taking the following steps:

1. Taking a SCC from the control flow graph
2. Making two copies of this SCC, named copy 1 and copy 2
    (a) Applying decomposition on copy 1 to produce several SCCs, and estimate total reduced cycles

   (b) Apply Knapsack Packing instead of Decomposition on the other copy, copy 2.

3. Commit the best performing copy from step 2-1 or 2-2. If an SCC is decomposed, regenerate all SCCs.

4. Repeat step 1-3 until every SCC (either introduced from partitioning or from original CFG) is all Knapsack-packed.

In next section, we shall talk about more details of a dynamic loading mechanism employed during evaluation of our design.

2.7 The Dynamic Loading Mechanism

This section focuses on the supporting mechanism of dynamically-loading code into scratch-pad memory.

For convenience, we define the term ‘knapsack’ as follows:

- A knapsack is a dynamically loaded code in 2.1
- A knapsack is effectively a knapsack-packed strongly-connected component in modified control flow graph computed using the algorithm defined in section 2.6.

That is, we call the dynamically-loaded code generated using our divide-and-conquer approach as ‘knapsacks’.

2.7.1 Assumption

Before we introduce our dynamic loading mechanism, a few assumption must be mentioned:

Assume all branch sources and targets are known at compile time.

Some cases in which above assumption may not hold are listed below:

- Use of function pointers
- Dynamically loaded libraries
- Interrupt handling code

One shall note that cases violating above assumptions can still keep the correctness, but requires extra techniques and may cost extra instructions and execution time during compile-time and run-time. Here we assume that, at least in our evaluation, above cases do not happen. Hence the following subsections describing our dynamic loading mechanism shall be based on above assumptions.

The dynamic loading mechanism can be roughly divided into two stages, which are compile-time actions and run-time actions, respectively. We start describing the first stage from the point after the set of knapsacks are determined, using the algorithm described in section 2.6.

2.7.2 Compilation Actions for Dynamically-loaded Code

We shall describe a mechanism supporting dynamically loading of code into scratch-pad memory in this section. This mechanism are used as the model in our evaluation stage.

After the decision of contents of every knapsacks, every branch involving these knapsacks (whatever they are branch targets or branch sources) requires special care to guarantee correctness. As described in section 2.7.1, we assume every pair of branch source and branch target are known at compile time.

The basic idea of this mechanism is to put the task of loading the knapsack in which the branch target resides in to the branch site. Under abovementioned assumptions, we can be sure about the knapsacks in which branch source/target reside (In the following text, we shall simply call them "source knapsack" and "target knapsack" respectively).

For these branch source/target pairs with different source/target kanpsacks, we insert a jump to the loading code of the target knapsack, as illustrated in Figure 2.4.

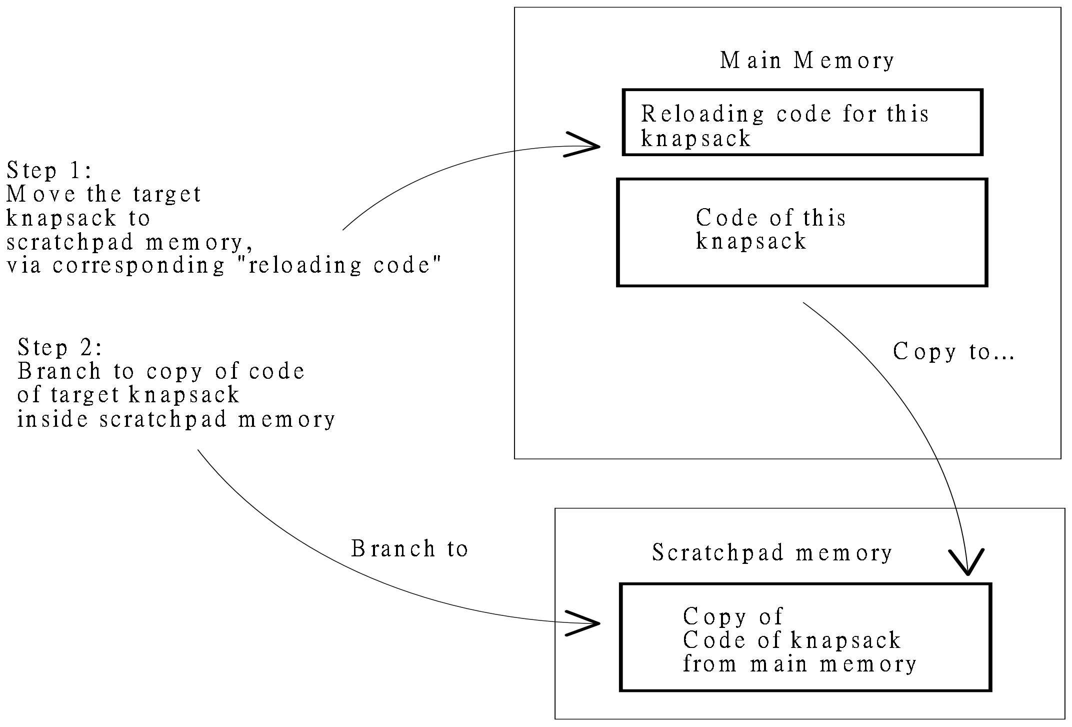
Main Memory
Reloading code for this knapsack
Code of this knapsack
Step 1: Move the target knapsack to scratchpad memory, via corresponding "reloading code"
Copy to...
Step 2: Branch to copy of code of target knapsack inside scratchpad memory
Branch to
Scratchpad memory
Copy of Code of knapsack from main memory

Figure 2.4: Dynamic Loading Mechanism

And, for the remainder case which branch source/target pairs belongs to the same kanpsacks or when the reloading is not required (e.g., branch source is from the code executed in main memory, and the content of scratch-pad memory already contains the target knapsack).

Chapter 3

Evaluation

In this chapter, we evaluate our proposed algorithm in the following aspects:

1. Time Complexity

2. Reduction of Problem Size

3. Quality of Solution

They are presented in detail in the rest of this chapter.

3.1 Time Complexity of Our Heuristic Method

In this section we briefly describe the asymptotic upper bound of the time complexity of our algorithm. Since our algorithm is based on iterative decomposition of SCC of control flow graph, and suppose there are E edges in the control flow graph, there will be at most E iterations of decompositions executed.

For each iteration of decompostion, our heuristics executes the following tasks:

1. Finding SCCs in the control flow graph

2. Knapsack-Packing each SCC

The first task have $\theta(V + E)$ if the Tarjan's method is applied, where V is the number of nodes in the control flow graph. The second step takes $\theta(SV)$ for each SCC. While there are at most V SCCs, an asymptotic upper bound of time complexity of this algorithm is $O(ESV^2)$. However, since there are total V nodes in control flow graph, suppose V_i is the number of nodes in ith SCC, then:

$$\sum_i V_i = V$$

Hence:

$$\sum_i SV_i = S\sum_i V_i = SV$$

which is the asymptotic upper bound of each iteration of decomposition. So a closer asymptotic upper bound should be: $O(ESV)$.

3.2 Reduction of Problem Size

Table 3.1 shows how our SCC decomposition algorithm partitions the problem under different scratch-pad memeory constraints, in terms of percent of basic blocks per SCC. It shows that our method can effectively reduce the problem size, hence there are opportunties for integrating ILP-based methods along with our decomposition method, as some sort of trade-off between solution time and quality.

3.3 Quality of Solution

In this section we analysis the quality of solution produced by our algorithm, at the following aspects:

1. Lower Bound of Solutions Produced by Our Method
2. Empirical Results

Table 3.1: Percent of Basic Blocks per SCC

Scratch-pad Memory Capacity	CJPEG	DJPEG	EPIC	UNEPIC
32K	32.95%	32.99%	33.10%	32.83%
16K	32.95%	32.99%	33.10%	32.83%
8K	32.95%	32.99%	33.10%	32.83%
4K	32.60%	24.35%	33.10%	31.95%
2K	10.30%	12.53%	12.84%	7.96%
1K	1.47%	0.92%	3.58%	6.95%
512Bytes	1.21%	0.71%	3.35%	1.40%
256Bytes	1.24%	0.69%	3.05%	1.11%
128Bytes	1.15%	0.58%	2.67%	0.74%
Average	16.31%	15.42%	17.54%	16.51%

Because our method partitions the problem only if the result after partition outperforms the result by just knapsack-packing, our method is guaranteed to outperform static knapsack packing method given the same profiling data. Hence, the lower bound of the solution produced our method is the static method.

In next section, we further analyse the performance of our methods by empirical means.

3.4 Empirical Results

In this section, the performance (in terms of execution time improvement) of following configurations are evaluated in different memory capacity constraints:

1. Cache

   (a) A Two-way Set-Associative Instruction Cache, with block size setting to 32 bytes

2. Scratch-pad Memory with the following packing methods:

   (a) Static Knapsack Packing of Program

   (b) Dynamic approach with our decomposition method for partitioning problems, and knapsack packing for 'solving' subproblems.

The experiment methodologies for above two scratch-pad memory packing approaches and the simulation of cache are illustrated in Figure 3.1.

The rest of this section is organized as follows: Subsection 3.4.1 shows the experiment setting. In subsection 3.4.2 we present the experiment results. Finally subsection 3.4.3 discusses the experiment results.

3.4.1 Experiment Settings

Benchmarks

The following benchmark programs are selected from the MediaBench [Lee et al., 1997] with the criterion of similar sizes (Program size between 16K-32K), and similar application (still image compression and decompression) to reflect the typical workload of a still-image capturing and compression/decompression environment commonly seen in embedded devices. These two descriptions are adopted from Lee et al. [1997]:

JPEG A standardized compression method for full-color and gray-scale images. JPEG is lossy, meaning that the output image is not exactly identical to the input image.

EPIC An image compression utility that is based on a wavelet decomposition and a combined run-length/Huffman entropy coder.

Table 3.2: Benchmark programs

Benchmark	Program Name	Description
JPEG	`cjpeg`	JPEG Encoder
	`djpeg`	JPEG Decoder
EPIC	`epic`	EPIC Encoder
	`unepic`	EPIC Decoder

Both benchmark programs consists of two programs: the encoder and the decoder, respectively. Hence total four programs are benchmarked, see Table 3.2 for details about these benchmark programs.

Experiment Environment

To approximate an typical embedded system, we employ the ARM7TDMI [Jaggar, 1996] as our target processor. The compiler is the one comes from ARM Developer Suite (ADS) 1.2. Also, since our simulation environment is trace-driven, as illustrated in Figure 3.1, our instruction trace is obtained from the `armsd` ARM symbolic debugger. Finally, to convert the trace to control flow graph, we wrote a tool to construct the control-flow graph of trace (as an approximation of actual control flow graph in source program) from instruction traces. The whole flow is presented in Figure 3.2.

The construction of “control flow graph” from trace is about to find the “basic blocks” in trace. Note these “basic blocks” are not equal to actual basic blocks in source program if the trace do not cover all possible execution paths in the executable. To find a “basic block”, we apply the following procedure:

1. Assume that every instruction trace address belong to a different “basic block.”

2. If two instruction addresses are consecutive in the instruction trace, add an edge

to the "basic blocks" these instruction belongs to.

3. Merge these consecutive "basic blocks" with single entry and single exit.

3.4.2 Experiment Results

Figure 3.3 to Figure 3.6 shows the experiment results of our divide-and-conquer approach , which is labeled as 'Our method' on these figures. We compare its result to the performances of: a two-way set associative instruction cache, and the static knapsack approach (labeled as 'Knapsack, Static').

Every figures lists the average cycle per instruction fetch (the label 'Avg. cycle per I.F' in y-axis in Figure 3.3 to Figure 3.6), while their x-axis are different scratch-pad memory capacity, ranging from 32 kilobytes to 128 bytes except for the case of cache. The performance of cache is obtained by applying the instruction trace on a instruction-cache simulator, with block size set to 32 bytes, and the cache size is set to be the same as the corresponding scratch-pad memory size.

In all four benchmarks, our method has the lowest average cycle per instruction fetch. Comparing to static knapsack packing approach, results from `cjpeg`, `djpeg` and `unepic` show our approach will obtain some advantages by allowing multiple hot-spots in these benchmark programs use the scratch-pad memory in turn.

For `epic`, our method and static knapsack packing almost match. It implies that even in smaller-sized scratch-pad memory, knapsack-packing methods still performs better than the decomposition method. After investigating the source of `epic`, we can find the cause of such phenomenon: there is only a major, non-nested loop in the EPIC encoder. That may be the reason why the decomposition does not work.

As a result, experiment shows that our approach best performed when given applications with multiple hot-spots and nested loops. With the collaboration of knapsack-packing and decomposition, whether there exists multiple hot-spots or not, appropriate strategy can be composed. In the case without explicit multiple hotspots, our method is likely to degenerate to the static knapsack-packing method.

3.4.3 Discussion

Our current design use a 'always-flushing' scheme for replacement in scratch-pad memory. It means any existing knapsacks in scratch-pad memory must be flushed before loading of any new knapsacks. This effectively forbids any co-existing of knapsacks. However, it may be beneficial if we allows two knapsacks to co-exist in the scratch-pad memory if free space of scratch-pad memory allowing us to do so.

In the following sections, we do an experiment to explore the possible benefit by construct an imaginary, ideal replacement mechanism for the scratch-pad memory.

Experiment on allowing co-existing knapsacks in scratch-pad memory

This experiment is to explore the upper-bound of possible benefit from allowing co-existing of knapsacks in scratch-pad memory if there is sufficient space in scratch-pad memory.

Before showing results, we present our assumptions in this experiment:

- An ideal replacement strategy that has access to entire program trace.
- No consideration of internal or external fragments caused by replacement.
- No maintaining overhead of space allocation and deallocation.

Above assumptions implies only the action of copy knapsacks between main memory and the scratch-pad memory are concerned. Thus, it can be viewed as an ideal model for us to explore the potential profit of ‘co-existing knapsacks’ for our divide-and-conquer approach.

Table 3.3 shows the percentage of eliminated unnecessary re-loadings, as long as the percentage of total execution cycles earned from these reloading, comparing to an always-flushing scheme.

The baseline of this comparison is based on the always-flushing scheme. The scratch-pad memory capacity ranges from 32 kilobytes to 128 bytes, in binary exponential descending order.

The result of this experiment shows, though at most 71% of reloadings are saved in `djpeg`, the contribution of such scheme to total execution time is still non-significant, as being just 3.8%. Except for `djpeg`,all benchmark programs here show at most 1.65% reduction in total execution cycles. One shall note that, above performance evaluation is base on an ideal model, rather than a realistic ‘co-existing’ mechanism. That means if a realistic co-existing mechanism is applied, with realistic overheads and costs introduced by the software implementing allocation algorithm, replacement algorithm, etc, such co-existing scheme is unlikely to be an attractive approach.

Table 3.3: Upper-bound of performance improvement using the 'co-existence' scheme

Scratch-pad Memory Capacity	Benchmark	32K	16K	8K	4K	2K	1K	512B	256B	128B
Re-loadings avoided, comparing to always-flushing scheme	CJPEG	0%	0%	0%	0%	0%	14.05%	8.57%	8.43%	0.06%
	DJPEG	0%	0%	0%	0%	0%	0.78%	0.26%	71.20%	5.78%
	EPIC	0%	0%	0%	0%	2.25%	0.31%	0%	0%	0%
	UNEPIC	0%	0%	0%	0%	0%	0%	5.75%	1.21%	0%
Saved Cycles comparing to always-flushing scheme	CJPEG	0%	0%	0%	0%	0%	0.95%	1.65%	0.44%	0%
	DJPEG	0%	0%	0%	0%	0%	0%	0.04%	3.80%	0.21%
	EPIC	0%	0%	0%	0%	0%	0%	0%	0%	0%
	UNEPIC	0%	0%	0%	0%	0%	0%	0.01%	0%	0%

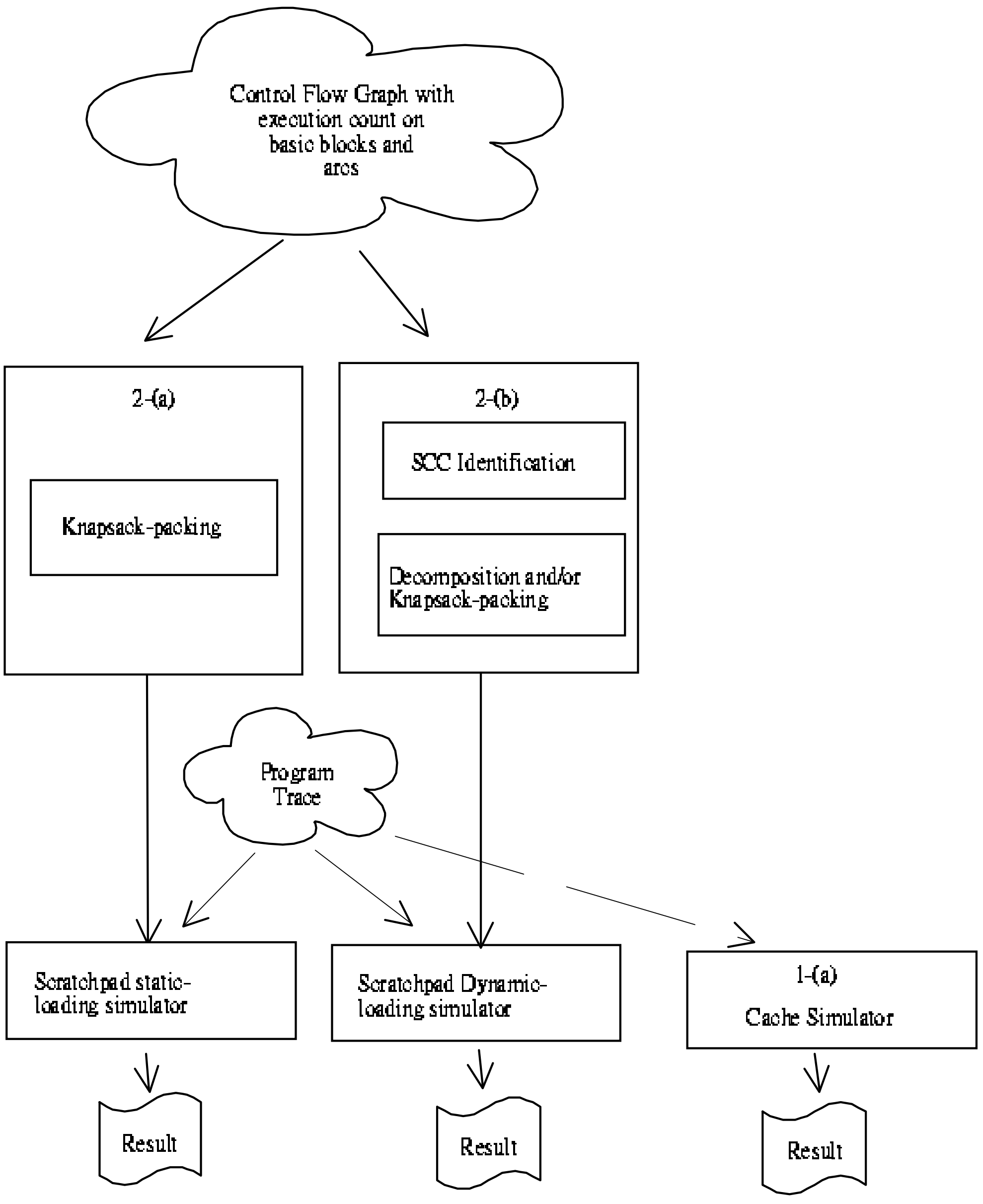
Control Flow Graph with execution count on basic blocks and arcs
2-(a)
Knapsack-packing
2-(b)
SCC Identification
Decomposition and/or Knapsack-packing
Program Trace
Scratchpad static-loading simulator
Scratchpad Dynamic-loading simulator
1-(a)
Cache Simulator
Result
Result
Result

Figure 3.1: The experiment flow of scratch-pad memory packing schemes and cache

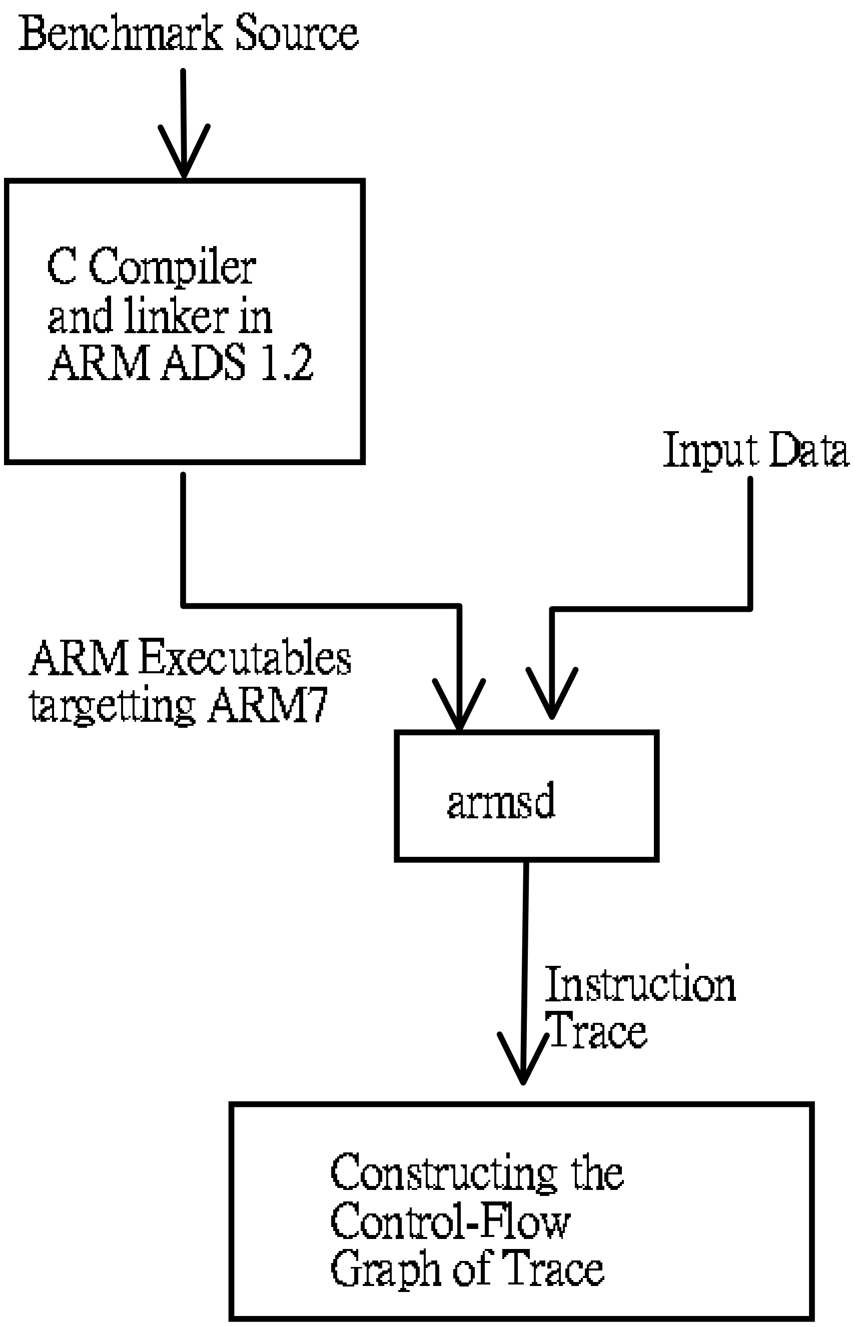
Benchmark Source
C Compiler and linker in ARM ADS 1.2
Input Data
ARM Executables targetting ARM7
armsd
Instruction Trace
Constructing the Control-Flow Graph of Trace

Figure 3.2: The Flow of Experiment: Obtaining “Control Flow Graph” of trace

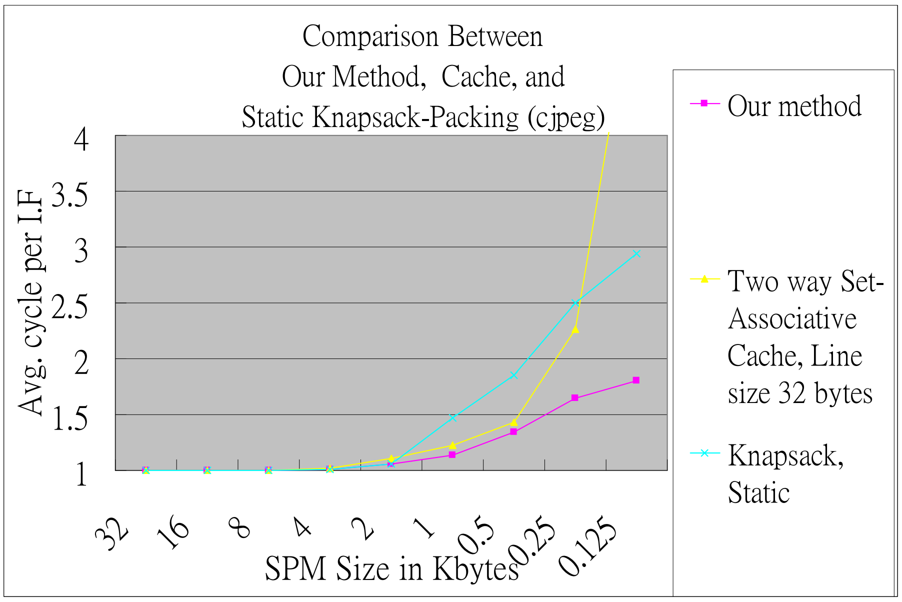

Figure 3.3: Comparisons among our methods, Static Knapsack-Packing, and Cache. Benchmark: cjpeg

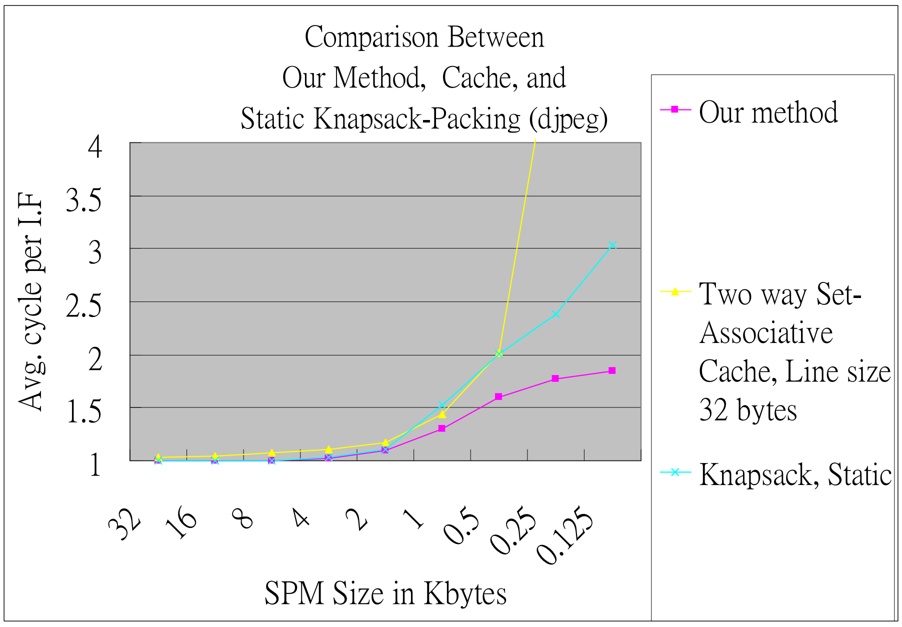

Figure 3.4: Comparisons among our methods, Static Knapsack-Packing, and Cache. Benchmark: djpeg

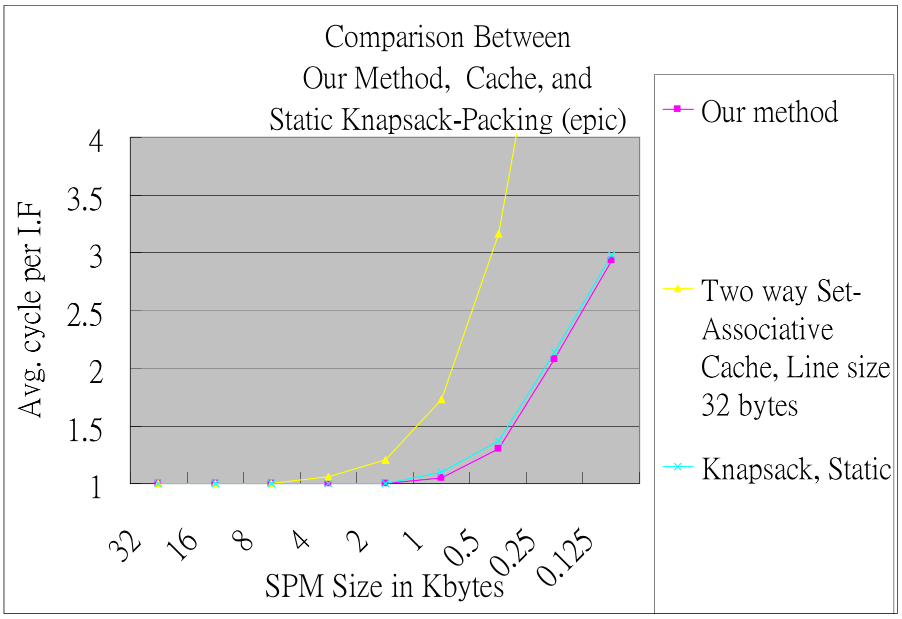

Figure 3.5: Comparisons among our methods, Static Knapsack-Packing, and Cache. Benchmark: epic

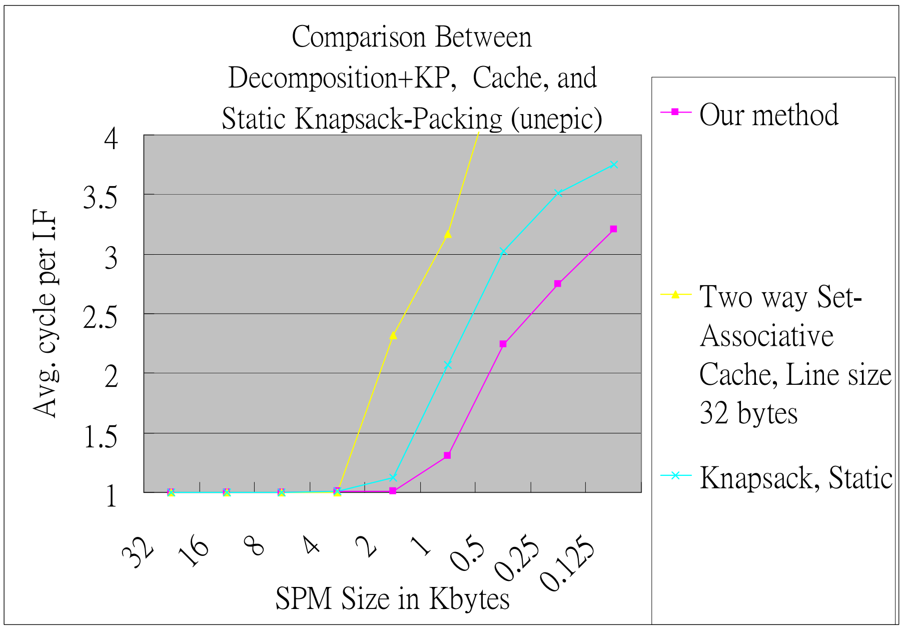

Figure 3.6: Comparisons among our methods, Static Knapsack-Packing, and Cache. Benchmark: unepic

Chapter 4

Conclusion and Future Directions

This chapter concludes this thesis. In Section 4.1 we summarize and conclude this study. Section 4.2 points out some possible issues worth further investigation.

4.1 Conclusion

In this thesis, a heuristic to solve the problem of dynamically loading program into scratch-pad memory is proposed and evaluated. This heuristic is based on the idea of 'divide-and-conquer'. Here 'divide' means the decomposition of nested loops, and 'conquer' means the one-dimensional 0/1 knapsack packing. Table 4.1 summaries the differences between our proposed method and previous methods.

The evaluation result is then compared to the static packing approach and a degeneration of above approach to evaluate the effectiveness of decomposition method. Result shows that, our approach performs the best with programs with multiple hot-spots and nested loops. The evaluation result also shows that, without hardware support, our software-controlled loading of program code can still outperform a two-way set-associative I-Cache.

Furthermore, experiment shows that a dynamic loading mechanism with the always-

Table 4.1: Comparison between Static, Dynamic [Steinke et al., 2001], and Our Approach

	Static Approach	Dynamic Approach	Our Approach
Problem Modeling	One-Dimensional 0/1 Knapsack Packing	Multi-Dimensional 0/1 Knapsack Packing	Heuristically-Partitioned Instances of 1-D 0/1 Knapsack Packing
Subject	Basic Blocks	Basic Blocks, packed in different loops	Basic Blocks, packed in SCCs in CFG
Solution	1-D 0/1 Knapsack Packing	ILP Solver	Heuristic Partitioning and 1-D 0/1 Knapsack Packing Static or Dynamic
Solution Quality	Optimal	Optimal	Suboptimal
Distingushing Feature	Packing all hot-spots into a single SPM context	Optimal dividing of SPM contexts and packing hot-spots into them through ILP framework	Divide SPM contexts by SCC decomposition, then independently pack each SPM context

flushing replacement strategy can be sufficiently close to the ideal case, in which multiple knapsacks can co-existing in scratch-pad memory if its capacity allows. Thus, with the simpleness of always-flushing, it will be the best replacement strategy for a dynamic loading mechanism to work in conjunction with our heuristic, divide-and-conquer based packing algorithm.

4.2 Future Directions

Still, there are some other issues worth further investigation, such as:

1. Take the interaction between knapsacks into consideration
2. Relax the constraint of strict partitioning of program parts between scratch-pad memory and main memory
3. Integrate this algorithm to a real compilation environment

The selection criteria of knapsacks to be dynamically loaded into scratch-pad memory is based solely on its estimated "earned" cycles. However, in some cases we may earn more cycles by not loading a knapsack which will introduce "threshing" in scratch-pad memory. That is, it is beneficial to consider the interaction between knapsacks along with their (worst-case) reduced cycles.

The second issue is to relax the modeling of "partitioning" (i.e. partition the set of basic blocks to two subsets: those who are executed in main memory, and those who are executed in scratch-pad memory). A knapsack can have different reduced cycles while dynamically-loaded into scratch-pad memory in differnt execution paths. Hence, a generalization in the problem modeling, instead of paritioning, is possible to save more cycles by making a knapsack to be dynamically-loaded into scratch-pad memory in one path, but executed in main memory in the other.

Finally, our study currently bases on an approximation of compilation environment. In the future we shall integrate this into a real compilation environment to further validate its performance in typical, real embedded systems.

References

R. Banakar, S. Steinke, M. Balakrishnan B. Lee, and P. Marwedel. Comparison of cache and scratch pad based memory system with respect to performance, area and energy consumption. In *Technical Report 762, University of Dortmund*, September 2001.

N. Hajj, C. D. Polychronopoulos, and G. Stamoulis. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):317–326, June 2000.

D. Jaggar. *ARM Architecture Reference Manual.* Prentice Hall, 1996.

C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *30th Annual International Symposium on Microarchitecture*, December 1997.

G.L. Nehmhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wilsey and Sons, New York, NY, 1988.

S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto on-chip memory, 2001. URL `citeseer.ist.psu.edu/562872.html`.

S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. of DATE, Paris, France*, March 2002.

Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.