Fall 1995

Identifying and exploiting concurrency in object-based real-time systems

Guohui Yu
New Jersey Institute of Technology

ABSTRACT

IDENTIFYING AND EXPLOITING CONCURRENCY IN OBJECT-BASED REAL-TIME SYSTEMS

by
Guohui Yu

The use of object-based mechanisms, i.e., abstract data types (ADTs), for constructing software systems can help to decrease development costs, increase understandability and increase maintainability. However, execution efficiency may be sacrificed due to the large number of procedure calls, and due to contention for shared ADTs in concurrent systems. Such inefficiencies are a concern in real-time applications that have stringent timing requirements. To address these issues, the potentially inefficient procedure calls are turned into a source of concurrency via asynchronous procedure calls (ARPCs), and contention for shared ADTS is reduced via ADT cloning. A framework for concurrency analysis in object-based systems is developed, and compiler techniques for identifying potential concurrency via ARPCs and cloning are introduced. Exploitation of the parallelizing compiler techniques is illustrated in the context of an incremental schedule construction algorithm that enhances concurrency incrementally so that feasible real-time schedules can be constructed. Experimental results show large speedup gains with these techniques. Additionally, experiments show that the concurrency enhancement techniques are often useful in constructing feasible schedules for hard real-time systems.

IDENTIFYING AND EXPLOITING CONCURRENCY IN OBJECT-BASED REAL-TIME SYSTEMS

by
Guohui Yu

**A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy**

Department of Computer and Information Science

January 1996

APPROVAL PAGE

IDENTIFYING AND EXPLOITING CONCURRENCY IN OBJECT-BASED REAL-TIME SYSTEMS

Guohui Yu

Dr. Lonnie R. Welch, Dissertation Advisor — Date
Professor of Computer and Information Science, NJIT

Dr. Dieter K. Hammer, Committee Member — Date
Professor of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands

Dr. Franz J. Kurfess, Committee Member — Date
Professor of Computer and Information Science, NJIT

Dr. James A.M. McHugh, Vice Chairman, Committee Member — Date
Professor of Computer and Information Science, NJIT

Dr. Peter A. Ng, Chairman, Committee Member — Date
Professor of Computer and Information Science, NJIT

Dr. Wilhelm Rossak, Committee Member — Date
Professor of Computer and Information Science, NJIT

Dr. Wei Zhao, Committee Member — Date
Professor of Computer Science, Texas A&M University

BIOGRAPHICAL SKETCH

Author: Guohui Yu

Degree: Doctor of Philosophy

Date: January 1996

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer and Information Science,
  New Jersey Institute of Technology, Newark, New Jersey, 1996
- Master of Science in Computer Science,
  Beijing University of Aeronautics and Astronautics, Beijing, P. R. China, 1986
- Bachelor of Science in Computer Science,
  Beijing University of Aeronautics and Astronautics, Beijing, P. R. China, 1984

Major: Computer Science

Journal Publications:

G. Yu and L. R. Welch, "A novel approach to off-line scheduling in real-time systems", *INFORMATICA Special Issue on Parallel and Distributed Real - Time Systems*, vol. 19, No.1, pp. 71-83, Feb. 1995.

L. R. Welch, G. Yu, J. Verhoosel, J. A. Haney, A. Samuel, and P. Ng, "Metrics for evaluating concurrency in reengineered complex systems", *Annals of Software Engineering*, Volume 1, 1995 (in press).

L. R. Welch, G. Yu, B. Ravindran, F. Kurfess, J. Henriques, M. Wilson, M. W. Masters, and A. Samuel, "Reverse Engineering of Computer-Based Control Systems", *International Journal of Software Engineering and Knowledge Engineering* (in press).

Conference Publications:

G. Yu, "Use of Concurrency Enhancement in Off-line Schedule Construction", *The 2nd Workshop on Parallel and Distributed Real-Time Systems*, pp. 32-37, IEEE, April 28-29, Cancun, Mexico 1994.

G. Yu and L. R. Welch, "Program Dependence Analysis for Concurrency Exploitation in Programs Composed of Abstract Data Type Modules", *Sixth IEEE Symposium on Parallel and Distributed Processing*, pp. 66-73, Dallas, Texas, October, 1994.

J.P.C. Verhoosel, G. Yu, L.R. Welch and D.K. Hammer, "Pre-Run-Time Scheduling for Object-Based, Concurrent, Real-Time Applications", *Proceedings of the 2nd IEEE Workshop on Real-Time Applications*, pp. 8-11, IEEE, July 1994.

A. D. Stoyenko, L. R. Welch, P. Laplante, T. J. Marlowe, C. Amaro, B. Cheng, A. K. Ganesh, M. Harelick, X. Jin, M. Younis, and G. Yu, "A Platform for Complex Real-Time Applications," *The Complex Systems Engineering Synthesis and Assessment Technology Workshop*, Naval Surface Warfare Center, July 1993.

G. Yu, L. R. Welch, W. Rossak, and A. D. Stoyenko, " Automatic Retrieval of Formally Specified Real-Time Software Components," *Fifth Annual Workshop on Software Reuse*, October 1992.

Conference Presentations:

G. Yu, "Use of Concurrency Enhancement in Off-line Schedule Construction", *The 2nd Workshop on Parallel and Distributed Real-Time Systems*, pp. 32-37, IEEE, April 28-29, Cancun, Mexico 1994.

G. Yu and L. R. Welch, "Program Dependence Analysis for Concurrency Exploitation in Programs Composed of Abstract Data Type Modules", *Sixth IEEE Symposium on Parallel and Distributed Processing*, pp. 66-73, Dallas, Texas, October, 1994.

G. Yu, L. R. Welch, A. D. Stoyenko and W. Rossak, "Managing Libraries of Formally Specified Real-Time Software Components," *Reuse Education Workshop*, September 1992.

Technical Reports:

G. Yu and L. R. Welch, "Program Dependence Analysis for Concurrency Exploitation in Programs Composed of Abstract Data Type Modules", *New Jersey Institute of Technology*, No. CIS-94-10, March, 1994.

G. Yu and L. R. Welch, "A Novel Approach to Off-line Scheduling in Real-Time Systems", *New Jersey Institute of Technology*, No. CIS-94-17, March, 1994.

G. Yu, "A Method for Evaluating Large Scale Information Systems", *Research Report*, No.18, Nomura Research Institute and Nomura Computer Cooperation, Tokyo, Japan, March 1988.

Professional Experience:

Teaching and Research Assistant (January '91–Present)
Department of Computer and Information Science, NJIT, Newark, New Jersey.

Research Assistant (Internship) (Summer 1994)
NEC Research Institute, Inc., Princeton, New Jersey.

System Engineer (January '86–January '91)
National Information Center of China, Beijing, China.

Visiting Scholar (September '87–April '88)
Nomura Research Institute and Nomura Computer Corp., Tokyo, Japan.

This thesis is dedicated to my families:
my wife Yuan Li, my son Daniel Sishang, and my daughter Joanne Sini.

ACKNOWLEDGMENT

First let me give my special thanks to my thesis advisor Lonnie Welch for the constant stimulating discussions and his effort on helping me finish this thesis and many publications during the past four years.

I would like to thank all the other committee members, Dieter Hammer, James McHugh, Peter Ng, and Wei Zhao for their many valuable suggestions.

I am also thankful to Jack Verhoosel, Klaus Ecker, Amr Zaky, Franz Kurfess, Alex Stoyenko, Tom Marlowe, and all the members of the Software Engineering Lab for their valuable comments on the numerous presentations of my work.

Finally, I am very grateful to my families for their support during the past years in many different ways. They have always been my source of energy and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

Figure **Page**

CHAPTER 1

INTRODUCTION

The complexity and cost of software development and maintenance increase very quickly with the size of the software system. This makes it difficult for software developers to keep pace with the increasing demand for new systems. One way to address this crisis is to reuse the results of previous development efforts [50, 64, 79]. Abstract data types (ADTs), which are supported by languages such as Ada, C++, Clu and Modula-2, provide a mechanism for abstraction, encapsulation, modularity, and layering. Therefore, ADTs are often used to develop reusable software components.

1.1 The Problem and Motivations

The use of object-based mechanisms like abstract data types (ADTs) can help to decrease development costs, increase understandability, and increase maintainability. However, it may also increase execution overhead, due to frequent procedure calls. Furthermore, an ADT is often used to manage more than one object and can become a bottleneck in concurrent systems. Such inefficiencies are a concern in real-time applications that have stringent timing requirements. In [69, 78, 77, 68], the use of parallelism via asynchronous remote procedure calls (ARPCs) and replication (cloning) of ADTs is presented as a means to address these potential inefficiencies. In this work, a framework for concurrency analysis in object-based systems is developed, and compiler techniques for identifying potential concurrency via ARPCs and cloning are introduced. Additionally, the usefulness of these techniques is illustrated in the context of an incremental schedule construction algorithm that enhances concurrency incrementally so that feasible schedules for hard real-time systems can be constructed.

There are four concerns that motivate this thesis:

- ADTs are often used for constructing reusable software components [58, 62, 64, 24].
- The inefficient execution of programs with ADTs is a concern for time-critical applications [22, 30, 77, 62, 59, 63, 60, 61, 63].
- The potential for concurrency in ADT-based systems is quite high via asynchronous remote procedure call and ADT cloning [65, 67, 68, 69, 70, 72].
- Automatic incremental parallelization via ARPCs and ADT cloning for hard real-time scheduling has not been thoroughly investigated prior to the inception of this research.

1.2 Overview of the Off-line Scheduling Approach

Timeliness is a major concern for hard real-time systems. Programs may not be necessarily efficient as long as all the deadlines are met. When concurrency is applied to help in real-time scheduling, full parallelization may not be necessary since more resources (processors) are needed to achieve it. The motivation is: if one processor can do the job, why use two?

This thesis presents a new approach for constructing schedules off-line for real-time applications composed of abstract data type (ADT) modules. This approach constructs an initial schedule based on sequential execution. The initial schedule is evaluated for feasibility, and possible improvement (if it is not feasible). If the schedule needs to be improved and can be improved, the critical path and a list of critical methods are identified. Candidates are evaluated by analyzing the overhead each may cause, and the amount of potential concurrency if parallelization is applied.

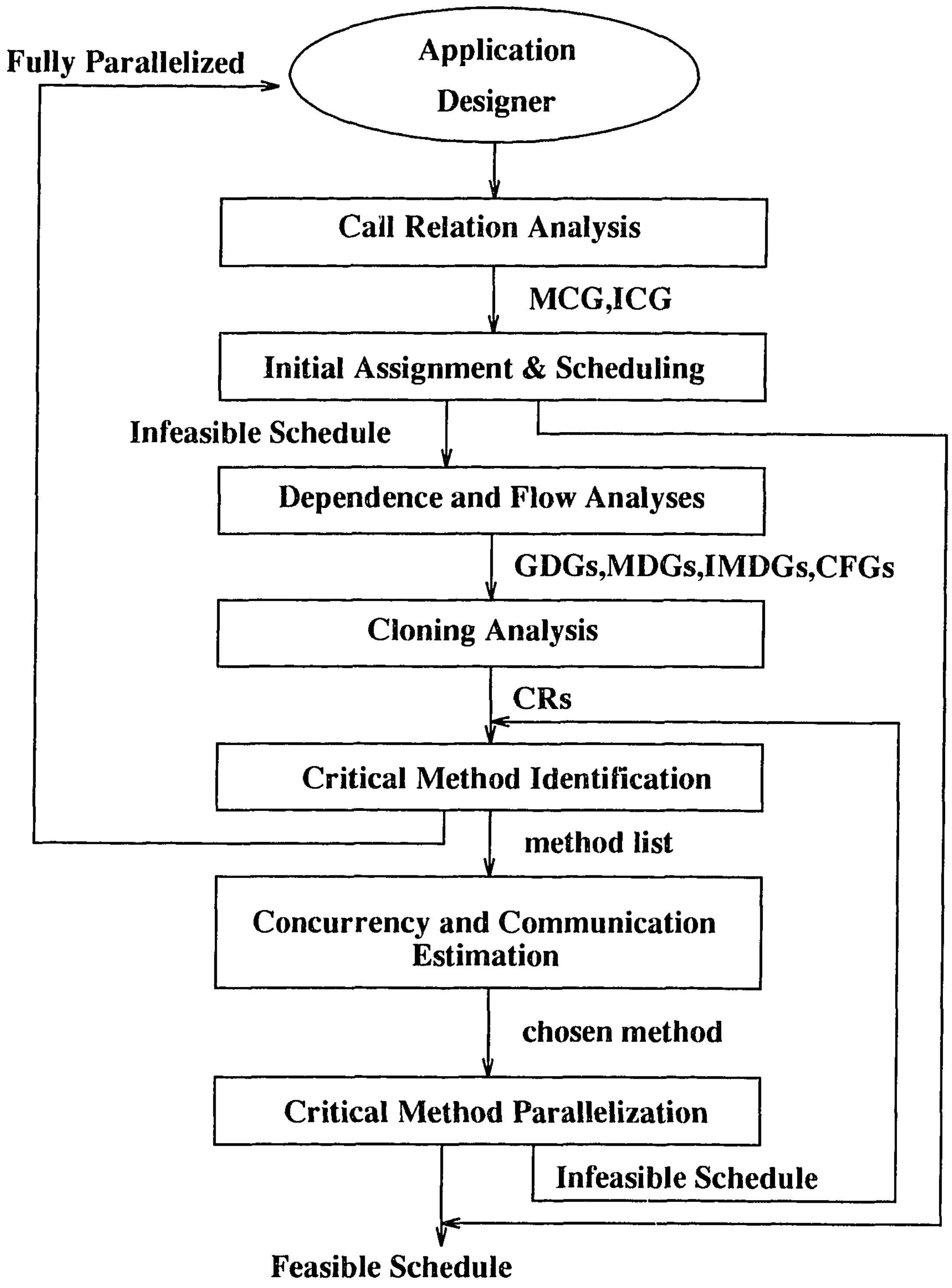


Figure 1.1 The scheduling approach.

The best candidate (possibly a combination of several candidates) is chosen to improve the schedule. The execution times of critical methods are reduced by exploiting and enhancing concurrency within the methods. The chance of finding a feasible schedule is significantly increased by concurrency enhancement. The overview of the off-line scheduling approach (shown in Figure 1.1) consists of the following steps:

1. The *Application designer* designs and implements an application using ADT modules.

2. *Call Relation Analysis* constructs the *method call graph (MCG)* and *instance call graph (ICG)* to describe the call relations among methods and ADT instances.

3. *Initial assignment and scheduling* generate an initial schedule based on sequential execution. The initial schedule is evaluated for feasibility. If the initial schedule is feasible, the process terminates.

4. *Dependence and flow analyses* construct a *control flow graph (CFG)*, a *general dependence graph (GDG)*, a *method dependence graph (MDG)*, and an *inter-method dependence graph (IMDG)* for each method. These dependence graphs describe various types of precedence relations among statements.

5. *Cloning analysis* determines the clone requirement (CR) of each method and instance needed to resolve all possible code contention.

6. *Critical methods identification* finds the critical path (the path that takes longest to execute) and the *critical methods* (the methods called by the statements on the critical path). These critical methods are analyzed for potential concurrency achieved and the overhead produced if parallelized.

7. *Concurrency and communication estimation* determines the communication cost a method can produce if it is parallelized. The method that has large ratio of concurrency to the overhead is determined. The amount of concurrency of a method is measured by the execution time reduced if the method is parallelized. The amount of overhead is measured by the number of processors needed and communication time needed.

8. *Critical method parallelization* parallelizes a critical method and updates the schedule. The parallelization of a critical method includes:

    - cloning the critical method to resolve contention;
    - using ARPCs to allow caller and callee to run concurrently;
    - reordering statements to enable ARPCs and cloning.

    The new schedule is evaluated. If it still infeasible, the infeasible schedule is sent back to the *Identifying Critical Methods* stage to be parallelized further.

This thesis presents solutions to each step of this scheduling approach.

1.3 Previous and Related Work

Parallelizing compiler techniques can be traced back to the 1960's with parallelization of FORTRAN programs [8]. The techniques were applied to all most every procedural programming language [26, 3, 2]. Those techniques were mostly for loops [33, 5, 42, 1, 38] and data structures [73, 10], and focused on fine grain of concurrency. Modern systems have become complex and large, and exploiting fine grain concurrency requires too much effort (time and space). Likewise, the properties of object orientation such as *abstraction, encapsulation, polymorphism, inheritance, layering*, and *modularity* are especially useful for developing large scale systems. Work in [81], techniques for automatical detection of intra- and inter-object parallelism for C++

programs. Unfortunately, parallelization of object-based systems has not been investigated thoroughly. This thesis presents techniques for identifying and exploiting concurrency for systems constructed with arbitrary data types – ADTs. Concurrency is gained at larger granularity: method level and instance level, and is used to meet "thread-level" objectives and constraints.

Automatic parallelization of programs depends on the dependence analysis of programs. Much has been published about control and data dependence analysis [8, 55, 32, 31, 19, 17, 4, 27]. Work in [8, 55, 32, 31] discuss various ways of dependence analysis techniques for parallelizing FORTRAN and assembly language programs. Work in [19, 17, 4, 13, 27] introduces the *program dependence graph (PDG)* and the continuation of PDGs to represent data and control dependence relations in a program. Those dependence relations determine the necessary sequencing between methods and can be used to expose potential concurrency. For example, if two calls have neither data nor control dependence, they are able to run in parallel.

Work in [56, 10, 9] uses interprocedural dependence analysis to determine whether or not the procedure calls prevent parallel code from being generated within a loop. Procedure calls that can be executed concurrently are detected.

Cloning has been applied previously, but in different contexts and for different objectives than those discussed here. Previous work on cloning is mainly concentrated on compiler optimization and fault tolerance. Keith Cooper [15] uses cloning techniques for compiler optimization. Clones of procedures are used to inherit an environment that allows for better code optimization. Procedure or task cloning for fault tolerance is discussed in [44, 11, 14]. They use clones of procedures or tasks to obtain high availability.

Cloning of ADTs for concurrency is first addressed in [69, 68, 78]. In [69], the asynchronous remote procedure call model and cloning are introduced, and a parallel virtual machine to support ARPCs and cloning is defined. In [68], the contention for an ADT is revealed by partitioning the statements of an ADT module into *units*, where a unit which is a sequence of statements that must be executed in order, due to data dependence. The approximate upper bound on the number of clones of an ADT instance that can be used concurrently is determined by a polynomial-time algorithm. The limitations of the work in [68] are: Clonability analysis is applied on ADT instance level only; Ignore the potential concurrency which may exists among statements across conditionals or loops; statement reordering for concurrency is not considered. This work is different from [68] in the following aspects:

- PDGs are extended to represent code dependence relations at three levels (statement level, method level, and instance level).
- The dependence and cloning analysis techniques can be applied at all three levels.
- Statement reordering for concurrency is discussed.
- Method cloning analysis and ways for handling conditionals allow more accurate upper bound on the number of clones of each ADT instance that can be used concurrently is determined.
- Also, the techniques are used for off-line schedule construction in hard real-time systems [76, 77], and are also used to help the U.S. Navy's reengineering efforts for mission critical systems [72, 71].

Most of the previous off-line scheduling techniques are searching algorithms for seeking feasible solutions by trying all possible permutations of processes, tasks, segments of programs [37, 49, 47, 74], or by using heuristics to guide their searching [30, 44, 61, 48]. The timing behavior of scheduling objects is unchanged during scheduling, thus all effort is devoted to optimizing the search path for finding feasible schedules. Work in [23, 29, 21, 22] applies compiler techniques to improve the performance of real-time programs so that feasible schedules can be found. In [23], a compiler classifies application code on the basis of its predictability and monotonicity, and creates partitions which have a higher degree of adaptability. Work in [29, 21, 22] applies code transformation and code motion to help tune real-time programs so that worst-case execution times are consistent with the real-time requirements. The performance of programs is improved by code transformation and code motion.

The rest of this thesis is organized as follows: Chapter 2 defines the object-based programming, the concurrent execution, and real-time scheduling models assumed in this research. Chapter 3 reviews the traditional dependence analysis techniques and introduces extensions for code dependence analysis. Chapter 4 introduces the cloning analysis techniques. Chapter 5 presents the incremental parallelization approach for constructing off-line schedules in hard real-time systems. The upper bounds obtained by cloning analysis are used as metrics to guide the incremental parallelization process. This off-line scheduling is done in conjunction with concurrency enhancement to improve the time behavior of tasks missing deadlines [77, 76]. The experimental results are presented in Chapter 6. Finally, the conclusions and open problems are given in Chapter 7.

CHAPTER 2

PROGRAMMING, EXECUTION, AND SCHEDULING MODELS

This chapter introduces the programming and execution paradigms assumed in this thesis. The generic abstract data type programming model is discussed first. This is followed by the introduction of the ARPCs and ADT cloning concurrency model. Finally, the real-time scheduling model is introduced,

2.1 Programming Model

It is assumed that software systems are composed of layered ADTs. A typical ADT module exports a type that can be used to declare variables, and operations to manipulate variables of the exported type. Although many work has been done on aliasing analysis [7, 16, 12, 34, 43], it is still impossible to determine which variables may be referenced by a pointer. In this programming model, aliasing is avoided by restricting the operations on pointers and by using swap parameter passing mode. As explained in [25], component efficiency increases when the values of composite data structures are swapped instead of copying them. Consequently, the parameter passing mechanism assumed here is *call-by-swap*, which exchanges (at least conceptually) the values of the formal parameters with the values of the actual parameters at the time of a call and upon return from a call. Consequently, the same argument can not appear more than once in the same call statement. Although this approach was originally advocated for developing efficient reusable components [25], it has the added benefit of simplifying parallelism extraction; variable synchronization is straightforward. Call-by-swap also has the desirable property that it does not make pointers available to users (as is done with call-by-reference). Thus, problems such as aliasing that arise when pointer types are provided do not occur.

Tailorable modules can be developed in languages permitting definition of generic modules (modules parameterized by types, by operations, or by other modules). To use a generic module, it is instantiated by fixing its parameters. Instantiation creates a module instance.

Another language feature is the automatic initialization of variables, as supported in C++ and RESOLVE [25]. With this technique, each type has an initialization operation that not only allocates storage, but also creates an initial value. Additionally, Extraction of parallelism is simplified when ADTs are used [50, 69]. Thus, the cloning techniques in this work are intended to apply to ADTs. Furthermore, programs can not have global variables. To make the programs analysible, no aliased variables or goto statements are allowed, unbounded loops and unbounded recursion are forbidden (to enable timing analysis), all memory allocations are done before runtime, the types of variables are determined statically, and instantiation of classes must be done statically. Those restrictions are acceptable for real-time systems since nondeterministic behavior cannot guarantee timeliness of real-time applications [51]. Thus, the scoping rules are straightforward. Another feature is that modules can not be instantiated dynamically. Thus, the number of ADT instances used in a program is known at link-time.

ADT programming languages provide mechanisms for abstraction, encapsulation, modularity, and layering. However, these mechanisms do not guarantee that the programs automatically have the above properties. Some design rules and principles must be followed. Work in [28] presents forty-one principles for constructing ADT components in Ada. We assume that components are mostly constructed according to these guidelines, but our analysis does not require strict adherence to all of them. The most important principles are summarized below: [1]

[1]Although those principles are discussed for Ada, they are general rules for developing ADT reusable components.

- Export a type so that abstract state is maintained in variables of that type, not in package instances.
- For each exported type, export initialization and finalization operations.
- If any exported operation has a precondition, export operations sufficient for a client to test that precondition.
- Do not export any exceptions, and design exported operations so that they do not raise any exceptions.
- Exported all types as limited private types.
- For each exported type, export a data movement operation (swap).
- Export and import (through generic formal parameters) all operations as procedures.
- Export a procedure that initializes all items declared internal to the package body that need initialization, e.g., variables, other package instances, etc.
- Export a procedure that finalizes all items declared internal to the package body that need finalization, e.g., variables, other package instances, etc.
- Parameterize the component by each ADT that it manipulates but does not export.
- For each type parameter to a generic package, import the type as limited private and import the type's standard operations: Initialize, Finalize, and Swap.
- In a client, use only the following constructs/statements: *block*, *case*, *exit*, *for loop*, *if*, *loop*, *procedure call*, *renames*, *return*, *while loop*.
- In any procedure call, do not use any variable as an actual parameter more than once.

- In any procedure call, do not supply as an actual parameter a variable that is globally known to the called procedure.
- Call an operation only if its precondition is satisfied.
- When implementing an operation in a component's package body, do not call any of the component's exported operations.
- In the package body of a component, do not declare local variables of the component's exported types at the package level, or in any of the component's exported or local operations.
- Add additional capabilities to a component by layering, when layering is possible.
- Directly implement an additional capability only if (a) layering is not possible or (b) the layered implementation has been shown to exact an unacceptable performance penalty for a particular application.

Figure 2.1 shows an example of an ADT module in Ada. It is a package taken from a virtual machine simulation application. It is a generic module which takes memory size and memory element type as parameters when an instance of the module is created. Thus, it can be used to create any size of memory for storing any type of variable. It exports a type called *MemoryType* and five operations as the interface of the module. The implementation of the type and the operations are hidden from users. Users can declare a variable using the exported type *MemoryType* and call the operations provided to manipulate the variable. Among the five operations, three of them are standard operations: *initialization*, *finalization*, and *swap*.

Figure 2.2 shows a portion of a factory simulation in RESOLVE. Figure 2.2 (a) shows a module *Queue* which is declared as a generic ADT module. The *Queue* takes a parameter *T* which is a type used to define the elements of the queue. The three

```
generic
MemorySize : in positive;
type itemType is limited private;
procedure initItem(item : out itemType);
procedure finItem(item : in itemType);
procedure swapItem(item1 : in out itemType; item2 : in out itemType);
package MemoryManager is
  type MemoryType is limited private;
  subtype indexType is Integer RANGE 1..MemorySize;
  procedure initialization(memory: in out MemoryType;
    item : in out itemType);
  procedure finalization(memory: in out MemoryType);
  procedure swap(memory: in out MemoryType,
    addr1 : in out indexType,
    addr2 : in out indexType);
  procedure fetch(memory : in out MemoryType;
    address : in out indexType;
    item : in out itemType);
  procedure store(memory: in out MemoryType;
    address : in out indexType;
    item : in out itemType);
  limited private
    type MemoryType is ARRAY(indexType) of itemType;
end MemoryManager;
```

Figure 2.1 An ADT module in Ada.

standard operations (*initItem*, *finItem*, and *swapItem*) are provided as parameters. When different types are provided, different queues can be instantiated from this ADT module. The ADT module *Queue* provides a type *QueueType* which can be used to declare queue variables such as *InQueue* and *OutQueue* in ADT module *Machine* (shown in Figure 2.2 (b)). Two methods *insert* and *remove* are provided by the *Queue* which is used to manipulate variables of *QueueType*. In Figure 2.2 (c), process *Task* is defined. An ADT instance M is created from module *Machine*, and is used to declare two machines $M1$ and $M2$. Figure 2.2 (c) shows only one statement of the *Task* which is a call to method $perform\_next()$ provided by the ADT instance M. Figure 2.2 (d) shows the main process which instantiates two processes, Task1 and Task2. Figure 2.3 shows a few call relations (edges) among three instances in the application.

```
module Queue(type T);
  provided type QueueType;
  facility ···
  procedure insert(var Q:QueueType;var x:T);
    begin
      ··· access Q and x for 5 time units
    end;
  procedure remove(var Q:QueueType; var x:T);
    begin
      ··· access Q and x for 5 time units
    end;
end Queue;
```

(a)

```
module Machine;
  provided type MachineType;
  facility part is Part(···);
  facility product is Product(···);
  facility Q is Queue(part.PartType);
  var inQueue, outQueue : Q.QueueType;
  var part : part.PartType;
  var prod : product.ProductType;
  procedure perform_next(var M:MachineType);
    begin
      ··· instructions for 2 time units
      Q.remove(inQueue,part);
      ··· access M, part, and prod for 3 time units
      Q.insert(outQueue,prod);
      ··· instructions for 2 time units
    end;
end;
```

(b)

```
process Task;
  facility M is Machine;
  var M1, M2 : M.MachineType;
  begin
    M.perform_next(M1);
    ···
end;
```

(c)

```
main process factory;
  process Task1 is Task
    with period=30, deadline=12,release-time=0;
  process Task2 is Task
    with period=60, deadline=21,release-time=0;
end;
```

(d)

Figure 2.2 Factory Simulation in RESOLVE

2.2 Concurrency Model

The execution platform used in this paper is a distributed memory MIMD system which is described in [69, 62]. Each processing element (PE) consists of a CPU, a communication co-processor and local memory. PEs are connected by either buses and/or high-speed, bidirectional point-to-point links. We assume that there is a physical route of buses and links between any pair of PEs.

Communication is handled by the communication co-processor in either a synchronous or in an asynchronous manner. Device resources are either *physical devices* or *logical devices.* Physical devices are hardware devices managed by software packages such as disks, sensors, and monitors. Logical devices are ADT module instances.

As shown in Figure 2.4, *execution graphs* are used to describe the executions of processes. In an execution graph, solid lines show executions of process segments, and dashed directed lines indicate calls and returns. The solid boxes indicate PEs, and the dashed ovals enclose operations of instances. EST stands for earliest start time, FT for finish time, and D for deadline of a process.

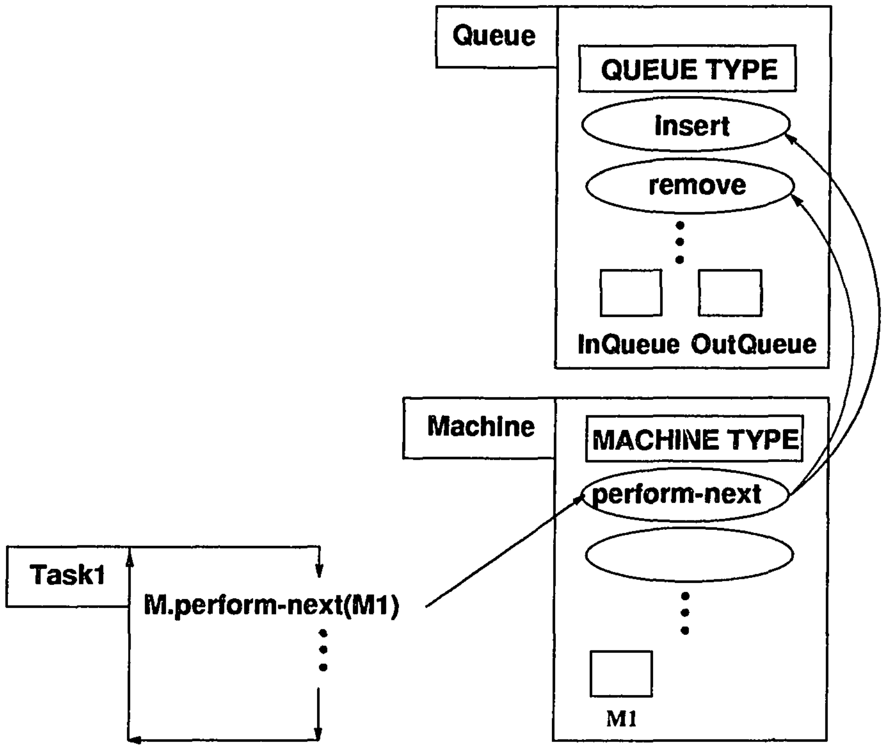


Figure 2.3 Instance relations created in factory application

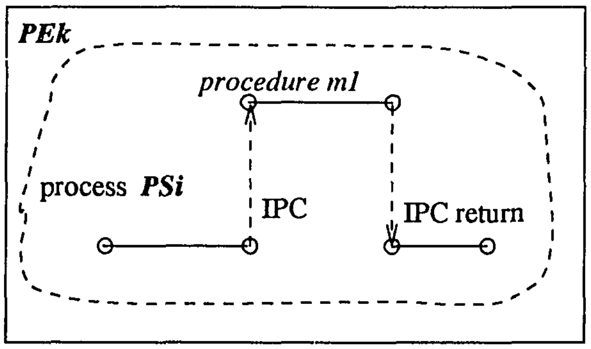


Figure 2.4 Internal Procedure Calls

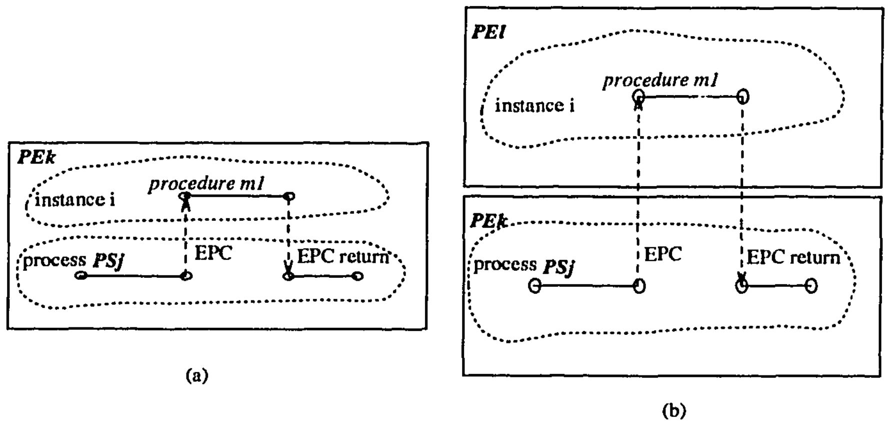


Figure 2.5 External Procedure Calls

An inter-instance call is an *internal procedure call* (IPC) if the calling operation is provided by an instance created within the activity class (as in Figure 2.4). For example, call $M.perform\_next(M1)$ in activity class $Task$ is an IPC since method $perform\_next()$ is provided by M which is instantiated within the activity class. An inter-instance call is an *external procedure call* (EPC) if the calling operation is provided by an instance created outside the activity class (as shown in Figure 2.5). Based on the physical location, IPCs and EPCs are implemented with *local procedure calls (LPCs)* and *remote procedure calls (RPCs).* An LPC occurs when the caller and the callee are on the same PE. An RPC occurs when the caller and callee are on two different PEs (as shown in Figure 2.5 (b)). An IPC is implemented by using an LPC which is simply a local context switch. An EPC is implemented by an LPC if the called operation is on the same PE (as shown in Figure 2.5 (a)), or by an RPC if the called operation is on a different PE (as in Figure 2.5 (b)). Processes are distributed among the PEs. A process and all of its ADT module instances are initially assigned to the same PE (as shown in Figure 2.6 for Task1 in the factory simulation).

The ADT methods and instances constituting a software system are distributed over the processors of the parallel computer and interact via remote procedure calls. Concurrency can be gained among processes if they are running on different PEs. RPCs can be *synchronous remote procedure calls (SRPCs)* or *asynchronous remote procedure calls (ARPCs)* [40, 35, 65, 66, 67, 70, 69]. With SRPC model, the caller is blocked to wait for the call to return. Concurrency is achieved among multiple processes by distributing them onto multiple PEs. With ARPC model, concurrency is gained by allowing a caller to continue execution until it requires a parameter passed by-reference to a remote operation that has not yet returned. When such a parameter is returned, the waiting caller is invoked to continue execution. Therefore, concurrency is achieved from not only statements in different PEs, but also statements (method calls) in the same PE. Initially, a process starts executing at one processor. The statements of the program are processed one by one; if a statement following an ARPC does not access any parameters passed to the ARPC, the statement is processed; otherwise it is blocked. Figure 2.7 shows four blocks b_1, b_2, b_3, and b_4. Blocks b_1 and b_3 are remote procedure calls which call the ADT operations on other processors. If remote procedure calls are handled synchronously, the execution of b_2 and b_4 are blocked until the return of b_1 and b_3, respectively (as shown in Figure 2.7 (a)). If remote procedure calls are handled asynchronously, and if the parameters used by b_1 and b_3 are used by b_4 only, b_2 can continue execution after b_1 is sent out since b_2 does not access the arguments used by b_1. Then b_3 can be sent to another processor since b_3 does not access the arguments used by b_1 either. But b_4 is blocked until the return of both b_1 and b_3 because it needs to access the data which might be modified by b_1 and b_3. Therefore, through ARPCs, b_1 and b_2 run concurrently and b_1 and b_3 run concurrently as shown in Figure 2.7 (b).

To control the complexity of scheduling, a hybrid scheduling model is used. Full preemption indicates that the execution of a process can be interrupted by other processes (with higher priorities) at any time. Nonpreemption describes a scheduling approach wherein a process must run to completion once it starts. Full preemption

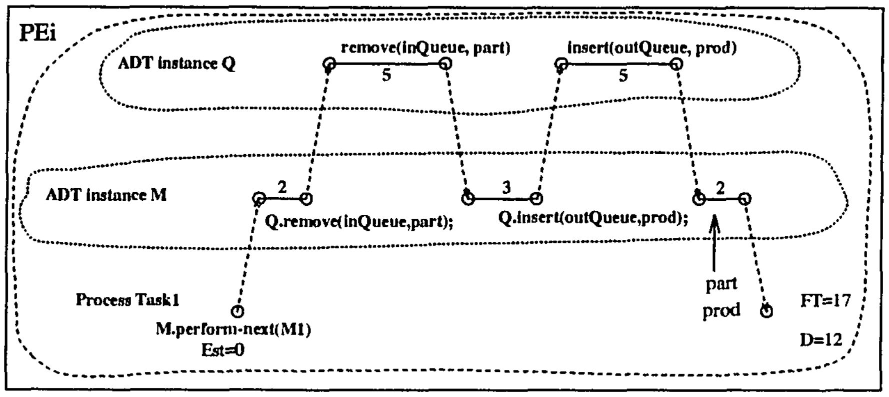


Figure 2.6 Assignment of Taskl and ADT instances in the factory simulation

gives the scheduler more flexibility to find a feasible schedule, but it increases the steps of scheduling because more cases are considered, and also may cause too much overhead (context switch) at run time. On the other hand, nonpreemption simplifies the scheduling job, but reduces the flexibility (chances to find a feasible schedule). To balance the flexibility and the complexity, we use a *semi-preemption* model [62] that restricts preemption to points called *preemption points.* When statements are used as scheduling units, there is maximal flexibility for scheduler and maximal overhead. Statements are grouped into segments called *basic block* (or simply called block) [20]. A basic block is a group of statements that have to be executed together without preemption. In this model, a basic block is a nonpreemptable scheduling and execution unit. Preemption points are the beginning and ending of basic blocks. Semi-preemption is a balance between the flexibility and the complexity of traditional scheduling approaches. More details about this execution model can be found in [62].

The ADT methods and instances constituting a software system are distributed over the processors of the parallel computer and interact via remote procedure calls. Concurrency is gained by allowing a caller to continue execution until it requires a parameter passed by-reference to a remote operation that has not yet returned. When such a parameter is returned, the halting caller is invoked to continue execution. This

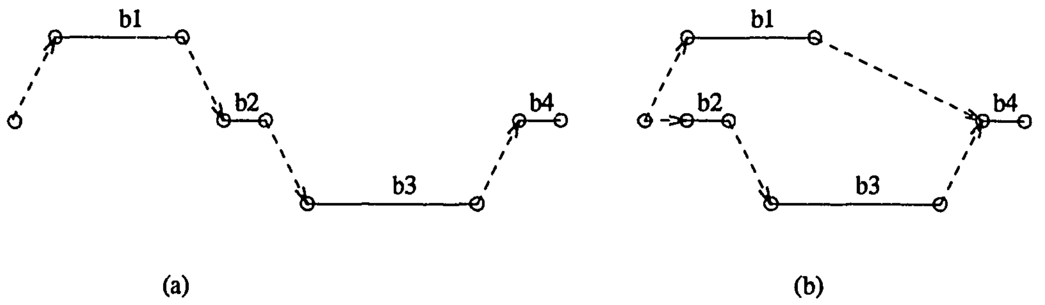


Figure 2.7 (a) SRPC. (b) ARPC.

model of concurrency execution is called *asynchronous remote procedure call (ARPC)* [40, 35, 65, 66, 67, 70, 69]. Intuitively, a sequential program starts at one processor. The statements of the program are processed one by one, if a statement following an ARPC does not access any parameters passed by reference to the ARPC, the statement is processed, otherwise blocked. Figure 2.7 shows four statements b_1, b_2, b_3, and b_4. Statements b_1 and b_3 are remote procedure calls which call the ADT operations on other processors. If remote procedure calls are handled synchronously, the execution of b_2 and b_4 are blocked until the return of b_1 and b_3, respectively (as shown in Figure 2.7 (a)). If remote procedure calls are handled asynchronously, and if the parameters used by b_1 and b_3 are used by b_4 only, b_2 can continue execution after b_1 is sent out since b_2 does not access the arguments used by b_1. Then b_3 can be sent to another processor since b_3 does not access the arguments used by b_1 either. But b_4 is blocked until the return of both b_1 and b_3 because it needs to access the data which might be modified by b_1 and b_3. Therefore, through ARPCs, b_1 and b_2 run concurrently and b_1 and b_3 run concurrently as shown in Figure 2.7 (b).

With the ARPC model, there are three factors that may block the execution:

1. the current statement is control dependent on the previous statement which is an ARPC. The control dependence describing the forced sequence of execution will block the execution of the current statement;

2. the current statement is data dependent on the previous statement which is an ARPC. The data dependence describing the exclusive access to the common data will block the execution of the current statement; and

3. the current statement is code dependent on the previous statement which is an ARPC, i.e., the two statements call the same operation.

The code dependence describing the exclusive access to the code of an ADT instance will block the execution of the current statement. Control dependence can be resolved by techniques such as concurrency execution of the multiple branches of if-statements and switch-statements, and concurrency execution for loop-statements [33, 5, 42, 6, 1, 38, 41]. Data dependence can be removed by data replication, variable renaming, and node splitting [41]. Code dependence can be resolved by code (executable) replication. In the programs built with ADT modules, some ADT instances may export operations that are heavily utilized and that become serialization points (bottle-neck) during parallel execution. Replication of codes can be applied at method level or instance level. The replicas (clones) of ADT methods or instances is distributed among the processors, allowing simultaneous use of the clones' code. As the example shown in Figure 2.8 (a), a call c_1 calls a method f and is being served. Before c_1 returns, another call c_2 arrives, it must wait until c_1 finish execution. If a clone of f is made and is placed in another processor, then c_1 and c_2 can run concurrently as shown in Figure 2.8 (b).

While cloning increases concurrency, it may increase CPU and network contention, and may also increase synchronization costs. These overheads vary for different kinds of ADTs. For cloning methods exported by stateless ADTs, there is no need to maintain consistency of data since no static data defined in stateless ADTs. For cloning methods exported by ADTs with states, additional effort on maintaining data consistency is needed. Therefore, the balance between the increased concurrency and the synchronization overhead is an important issue.

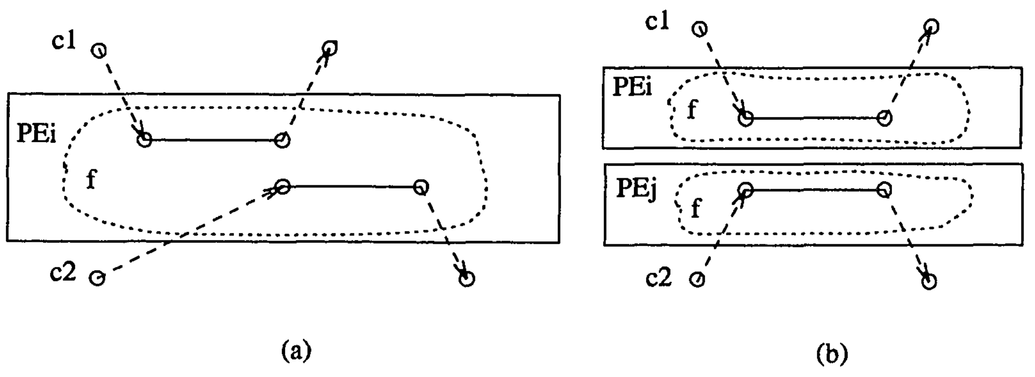


Figure 2.8 (a) Two calls are serviced sequentially by f. (b) Two calls are serviced concurrently by clones of f.

In this work, only methods exported by stateless ADTs are considered for cloning. cloning of ADTs with states is addressed as future work in Chapter 7. Note that cloning of methods allows granularity of concurrency at the method level, not at statement level. Therefore, communication costs are typically much smaller than the execution time of methods, especially since only pointers to data structures need to be passed in most remote procedure calls [65, 54, 69].

2.3 Scheduling Model

The job of scheduling is to decide the start times of scheduling objects so that their timing constraints are satisfied. As mentioned earlier, the general scheduling problem on multiprocessors is NP-hard. Optimal solutions [75, 49, 37] are not practical for large applications. Therefore, a heuristic approach is used to construct a schedule [76, 63]. The goal of scheduling is to resolve contention for shared resources. Generally, there are two kinds of shared resources: hardware resources (CPUs, I/O devices, and communication media) and software resources (ADT module instances). One way to resolve contention for shared resources is by replication of resources, as a multiprocessor system is used to resolve contention for the single CPU in an uniprocessor system. In the extreme, if the number of resources allows every client to get a resource at any time, there is no contention and there is minimal need for scheduling. However, the quantity of hardware resources is typically fixed in computing systems. Therefore, scheduling of hardware resources is necessary. Similarly, replication of software resources is also a way to resolve contention. In programs built by layering ADT module instances, an instance is often used to manage several data objects, and there will be contention for getting access to the instance if multiple data objects need to be accessed concurrently by multiple clients. Cloning an ADT instance allows each clone to manage only one data object, or a subset of the data objects.

Typically, systems constructed with ADTs have many layers and tend to have many method calls. Those method calls may be the source of concurrency via ARPCs and cloning. To improve a schedule by parallelizing programs, there are two approaches: top-down and bottom-up. The top-down approach parallelizes programs quickly since the execution times of methods at higher layers are longer than at lower levels. Therefore, parallelizing methods at higher layers produces more concurrency, but consumes more resources. With a bottom-up approach, methods at lower layers (with little resource requirements) are parallelized. However, a change made at a lower layer can affect all the methods and tasks from that layer up to the very top level. Thus, much effort is needed to update the schedule with a small change at bottom level. Thus, in this work, a top-down approach is used.

In the following subsections, several ways to enhance concurrency are presented. The example in Figure 2.2 is used to show how each kind of concurrency enhancement works in conjunction with the scheduling approach. Figure 2.2 (a) shows a module class *Queue* which is declared as a generic ADT module. The *Queue* takes a parameter T which is a type used to define the elements of the queue. When different types are provided, different queues can be instantiated from this ADT module. The ADT module *Queue* provides a type *QueueType* which can be used to declare queue variables such as *InQueue* and *OutQueue* in ADT module *Machine*

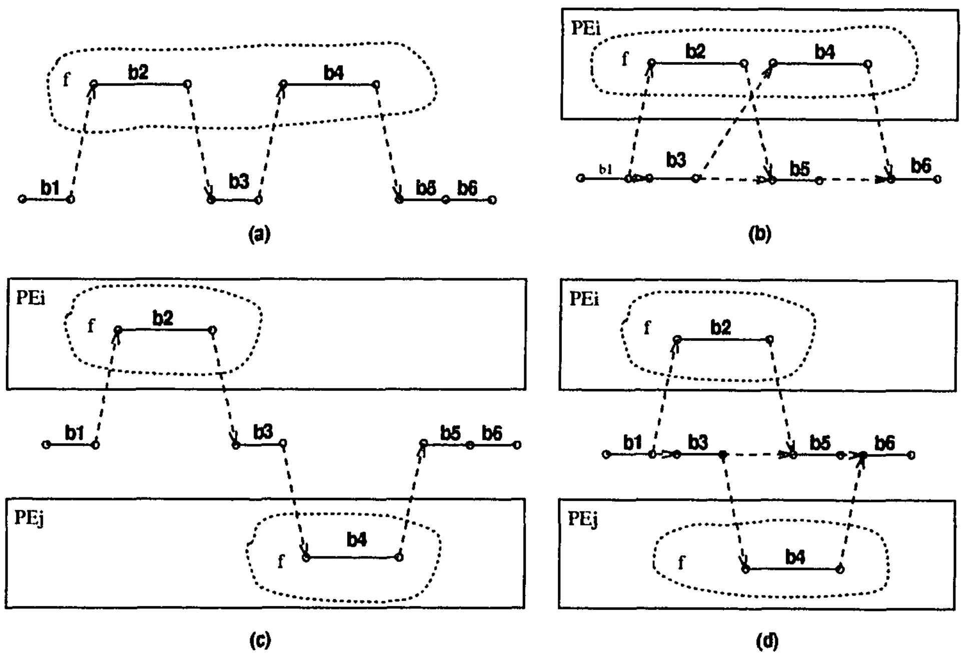


Figure 2.9 (a) Neither ARPCs nor cloning. (b) ARPCs only. (c) Cloning only. (d) ARPCs and cloning.

(shown in Figure 2.2 (b)). Two methods (*insert* and *remove*) are provided by the *Queue*, which is used to manipulate variables of *QueueType*.

2.3.1 Enhancing Concurrency via ARPCs

One way to introduce concurrency is to use ARPCs instead of SRPCs. With SRPCs, the caller is blocked after making a remote procedure call. Most of the scheduling approaches switch the calling processor to another process, but the calling process is blocked until the call returns. To reduce the execution time of the calling process, ARPCs can be used to let the caller continue execution if no memory conflict is caused. In Figure 2.9 (a), all calls are SRPCs and no concurrency exists. If instance f is allocated on another PE, and parameters used by statement $b2$ are not used until statement $b5$, and the parameters used by statement $b4$ are not used until statement $b6$, with the use of ARPCs, statements $b3$ and $b5$ can run concurrently with statements $b2$ and $b4$, respectively, and concurrency is achieved (as shown in Figure 2.9 (b)).

ARPCs make the calling method and called method run concurrently if they do not access the same memory location at the same time. By looking at the dependence relations among method calls, the opportunities for applying ARPCs can be identified [78]. In our scheduling approach, ARPCs are also used to reduce the execution times of processes missing deadlines.

2.3.2 Enhancing Concurrency via ADT Cloning

Cloning of resources can reduce contention for resources. If the number of clones of software resources in every processor is sufficient, then no contention exists. The question is, "How many clones of a resource is enough?" Another important question is "How many clones are needed to enhance concurrency to enable a schedule to meet deadlines?". To determine the lower bound on the number of clones needed, program dependence relations are analyzed. In [78], techniques are presented for determining the lower bound on the number of clones of each ADT module instance needed to resolve all possible contention of the ADT module instances. The technique employs dependence analysis techniques at the statement, method, and instance levels of granularities. The program dependence graph (PDG), which was previously used to describe data and control dependence relations among statements, is extended to include instance dependence relations in object-based systems. Several theorems are proved with respect to the instance dependence properties of the new PDG graph in [78]. In Figure 2.9 (c), ADT instance f is cloned and is placed on a different PE. Now two clones of instance f can serve two calls at the same time.

2.3.3 Enhancing Concurrency via ARPCs and ADT Cloning

If two calls do not access the same memory location at any time, but they call the same method or different methods provided by the same ADT module instance, the two cannot run concurrently since they have contention to access same ADT module instance. Cloning can be used to resolve the contention, and the SRPC can be converted into an ARPC. In the example in Figure 2.9 (d), ARPCs are combined with instance cloning so that maximum concurrency is achieved.

For the application in Figure 2.2, there exists no feasible schedule (no matter how process and module instances are assigned and scheduled) if concurrency enhancement is not applied. Assume an initial schedule is constructed as shown in Figure 2.6. Since all instances are in the same PE, all calls are local procedure calls. Therefore, only one thread of execution exists, i.e., the schedule of PE_i contains only one statement. Given such a schedule, our approach is to try to improve the schedule by enhancing concurrency. By examining the execution graph of the process $Task1$, we see that *Queue* is shared by two calls. Only one call is granted access to the *Queue* at any time; the other call is put into waiting. We also see that the two calls $Q.remove(inQueue, part)$ and $Q.insert(outQueue, prod)$ do not have common parameters, and do not call the same method, but they call the methods provided by the same ADT module instance Q. The two calls cannot run concurrently due to the contention for the instance Q. Cloning of Q can resolve the contention and turn the SRPC to an ARPC. In Figure 2.10, ADT instance *Queue* is cloned and placed on a different PE, and the two calls ($Queue.insert()$ and $Queue.remove()$) made by instance M can run concurrently. The one thread of execution in Figure 2.6 is broken into 4 statements. Blocks $b2$ and $b3$ run concurrently. The execution time of $Task1$ is reduced from 17 to 13. Therefore, concurrency is enhanced and an almost feasible schedule is found by using cloning and ARPCs, as shown in Figure 2.10.

2.3.4 Load Distribution to Allow ARPCs and ADT Cloning

A good schedule has high utilization of resources. If one process is unable to continue execution, the resource is given to another process. As we mentioned before, if the

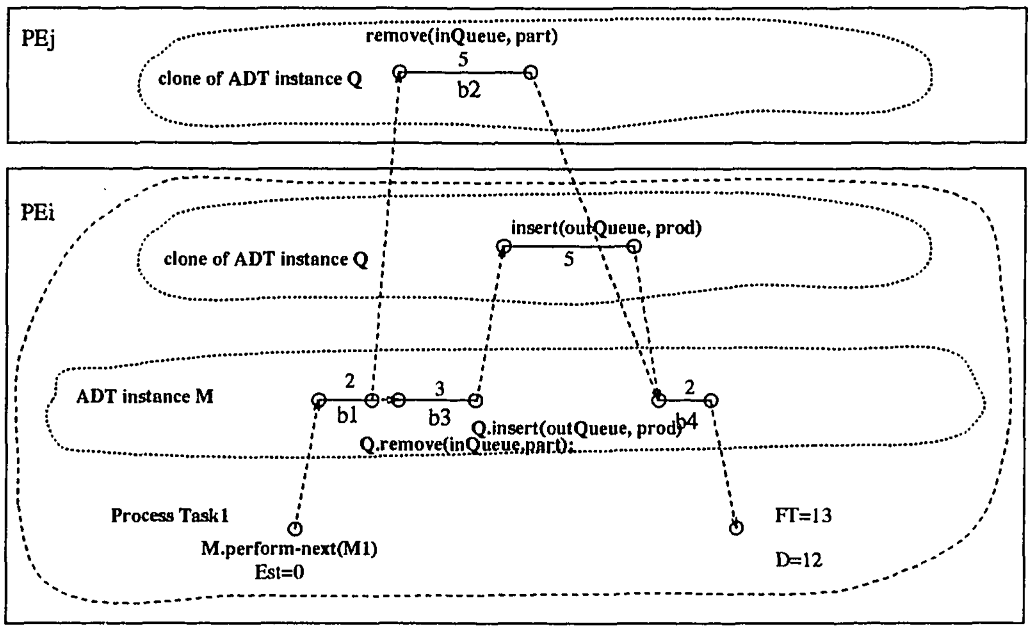


Figure 2.10 Two clones of Queue serve two calls concurrently, so that an almost feasible schedule is constructed.

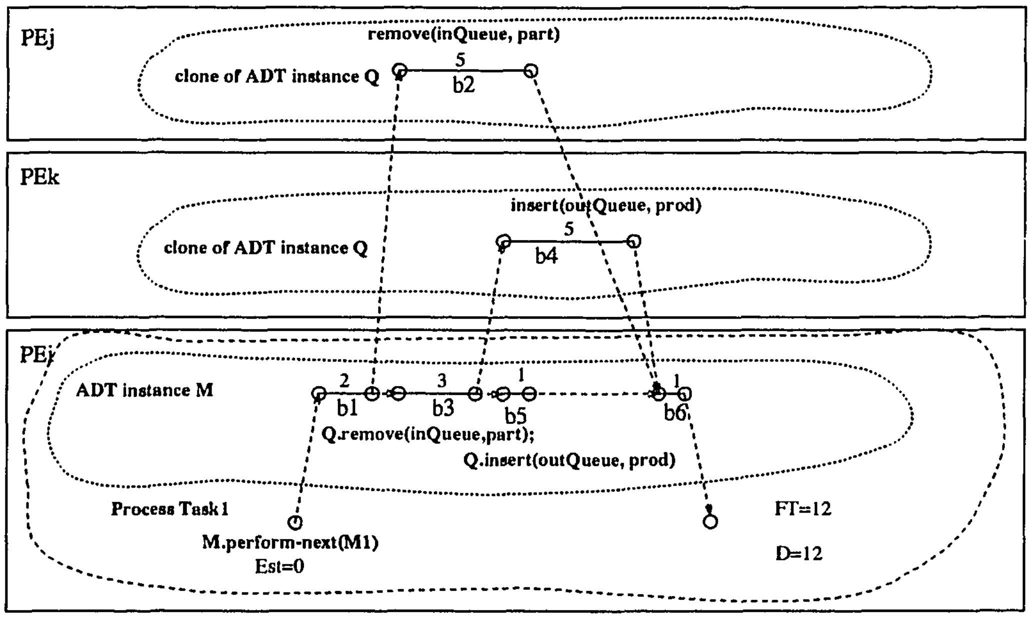
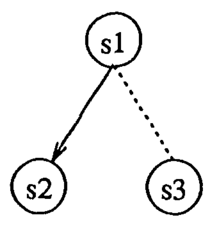


Figure 2.11 Enhancing Concurrency by Load Distribution.

```
s1    f.op1(x);
s2    g.op1(x);
s3    f.op1(y);
```

(a)



(b)

Figure 2.12 (a) A program segment. (b) The PDG of the program.

utilization of a PE reaches 100 percent, no ARPCs and cloning can be applied. To enable ARPCs and cloning, load distribution is necessary. In the example shown in Figure 2.2, note that the deadline of process $Task1$ is 12 time units, but its finishing time is 13 time units. The schedule in Figure 2.10 needs to be further improved . Although an ARPC opportunity exists (the first part of $b4$ will not access variables *OutQueue* and *prod* which are the parameters used by t he call *insert*()), PE_i is scheduled to execute method call *insert*(*OutQueue*, *prod*). If the ADT instance Q on PE_i is placed on PE_k as shown in Figure 2.11, block $b4$ can be an ARPC and can run concurrently with block $b5$. Thus, the finishing time of the task is reduced to 12 time units, and process $Task1$ can meet its deadline due to the load distribution.

2.3.5 Statement Reordering to Expose Concurrency

Although the upper bound of clones of an instance tells the maximum number of clones that can run concurrently, the statement order might prevent part of the concurrency. Let us see a simple example in Figure 2.12 (a). From its extended PDG [78] in Figure 2.12 (b), we can see that the maximum number of clones of instance f is 2, and the maximum number of clones of instance g is 1. However, the statement order s_1, s_2, s_3 prevents statements s_1 and s_3 from being executed in parallel, since statement s_2 would be blocked due to the busy parameter x until statement s_1 completes. The schedule is shown in Figure 2.13.

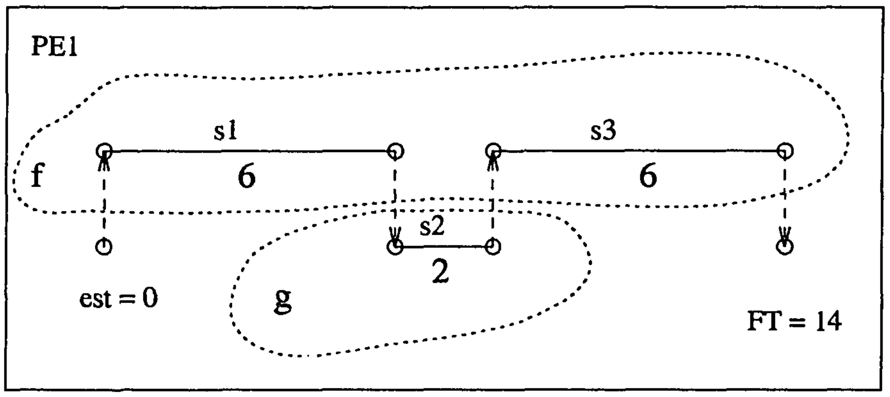


Figure 2.13 *Statement S*2 blocks the execution of *Statement S*3.

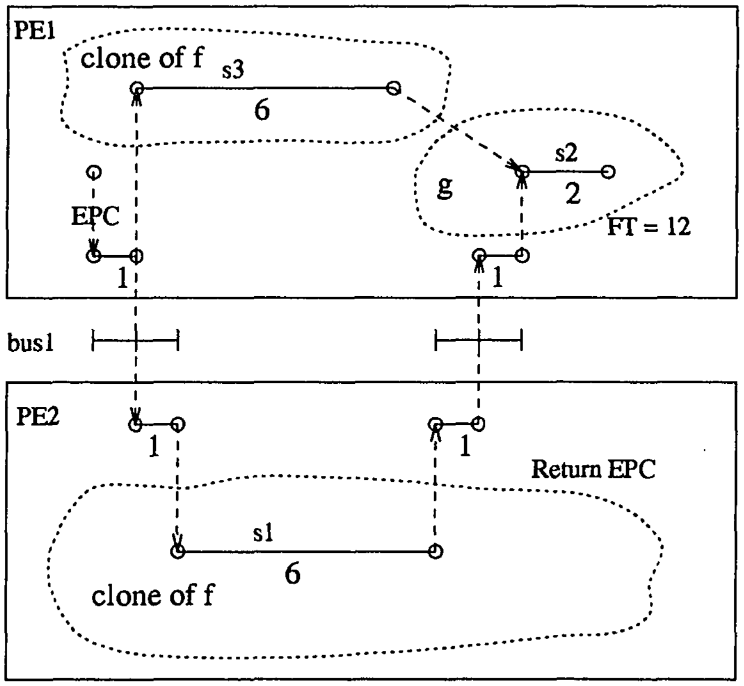


Figure 2.14 *Statement s*1 and *statement s*3 run concurrently by reordering *statement s*2 and *statement s*3.

If we reorder the statements as (s_1, s_3, s_2) or (s_3, s_1, s_2), statements s_1 and s_3 can be executed concurrently (allow two clones of instance f to be used in parallel). For the schedule shown in Figure 2.13, if we swap the two statements $s2$ and $s3$ and place a clone of instance f on $PE2$, the execution time of this program segment can be reduced from 14 time units to 12 time units, as shown in Figure 2.14.

CHAPTER 3

DEPENDENCE ANALYSIS

In order to identify and exploit concurrency, the dependence relations among statements, methods, and ADT instances must be analyzed. In this chapter, a brief review of program dependence graphs (PDGs) [19] is given in Section 3.1. This is followed by new algorithms for constructing the dependence graphs in Section 3.2. The extension of PDGs to describe code dependence relations is presented in Section 3.3. Some of the code dependence relations may falsely describe code contention due to the effect of precedence relations, thus, algorithms for identifying and removing such false or *ineffective* code dependence relations are presented in Section 3.4. Additionally, important properties of the extended PDGs are discussed and proved.

3.1 Traditional Program Dependence Analysis

In this section, the program dependence graph (PDG), which represents both control and data dependence relations among statements of a program is reviewed.

A virtual machine simulation program is used to illustrate the dependence graph concepts. In the virtual machine simulation, a generic memory management package (shown in Figure 3.1 (a)) is defined, and is used (in Figure 3.1 (b)) to instantiate two instances: *DataMem* for representing data memory and *InstrMem* for representing instruction memory. An ADT package *Processor* (shown in Figure 3.1 (b)) is defined to manage each processing element. For illustration, only the specifications of packages are shown, and only three processors are used. The main procedure *VM_Simu* (shown in Figure 3.1 (c)) is defined to coordinate (initialize and schedule) the processors.

```
generic
MemorySize : in positive;
type itemType is limited private;
procedure initItem(item : out itemType);
procedure finItem(item : in itemType);
procedure swapItem(item1 : in out itemType; item2 : in out itemType);
package MemoryManager is
 type MemoryType is limited private;
 subtype indexType is Integer RANGE 1..MemorySize;
 procedure initialization(memory: in out MemoryType;
  item : in out itemType);
 procedure finalization(memory: in out MemoryType);
 procedure swap(memory: in out MemoryType,
  addr1 : in out indexType,
  addr2 : in out indexType);
 procedure fetch(memory : in out MemoryType;
  address : in out indexType;
  item : in out itemType);
 procedure store(memory: in out MemoryType;
  address : in out indexType;
  item : in out itemType);
 limited private
  type MemoryType is ARRAY(indexType) of itemType;
end MemoryManager;
```

(a)

```
With MemoryManager;
package InstrMem is new MemoryManager(MemorySize=>50,
 itemType => String(1..10 => " ",
 initItem => String.init,
 finItem => String.fin,
 swapItem => String.swap));
With MemoryManager;
package Datamem is new MemoryManager(MemorySize=>50,
 itemType => Integer
 initItem => Integer.init,
 finItem => Integer.fin,
 swapItem => Integer.swap));
with Text_io, integer_io;
use Text_io, integer_io;
with DataMem, InstrMem;
package Processor is
 type PE_Type is limited private;
 type PEState_type is (IDLE, BUSY, WAIT);
 procedure initialization (PE : in out PE_type);
 procedure finalization (PE : in out PE_type);
 procedure doInstruction ( PE : in out PE_type);
 procedure getState(PE:in out PE_type;state:out PEState_type);
 procedure dumpDataMemory ( PE : in out PE_type,
  outfile : out File_type );
 private
  type PE_type is
   record
    State : PEState_type;
    Datam : DataMem.MemoryType;
    Instm : InstrMem.MemoryType;
    pc : Integer;
    instrReg : String(1..10);
    AC : Integer;
   end record;
end Processor;
```

(b)

```
      with TEXT_IO, file_support, integer_io;
      use TEXT_IO, file_support, integer_io;
      with Processor;
      procedure VM_Simu is
       PEA : ARRAY(1..NumPE) of processor.PE_Type;
       CurState : integer; - Current state of PE
       CurState1 : integer; - Current state of PE
       CurState2 : integer; - Current state of PE
       PECount : integer; - Temporary Variable
       n, pos1 : integer; - Temporary Variable
       outFile : String;
       begin
S1      PECount := 0;
S2      processor.initialization(PEA[1]);
S3      processor.initialization(PEA[2]);
S4      processor.initialization(PEA[3]);
S5      while (PECount < 3) loop
S6       PECount := 1;
S7       processor.GetState(PEA[1],CurState1);
S8       if (CurState1 == Idle) then
S9        processor.dumpDataMemory(PEA[1], outFile);
S10       PECount = PECount + 1;
         else
S11       processor.do_instruction(PEA[1]);
S12      processor.GetState(PEA[2],CurState2);
S13      if (CurState2 == Idle) then
S14       processor.dumpDataMemory(PEA[2], outFile);
S15       PECount = PECount + 1;
         else
S16       processor.do_instruction(PEA[2]);
S17      processor.GetState(PEA[3],CurState);
S18      if (CurState == Idle) then
S19       processor.dumpDataMemory(PEA[3], outFile);
S20       PECount = PECount + 1;
         else
S21       processor.do_instruction(PEA[3]);
        end loop;
       end VM_Simu;
```

(c)

Figure 3.1 The virtual machine simulation program in Ada.

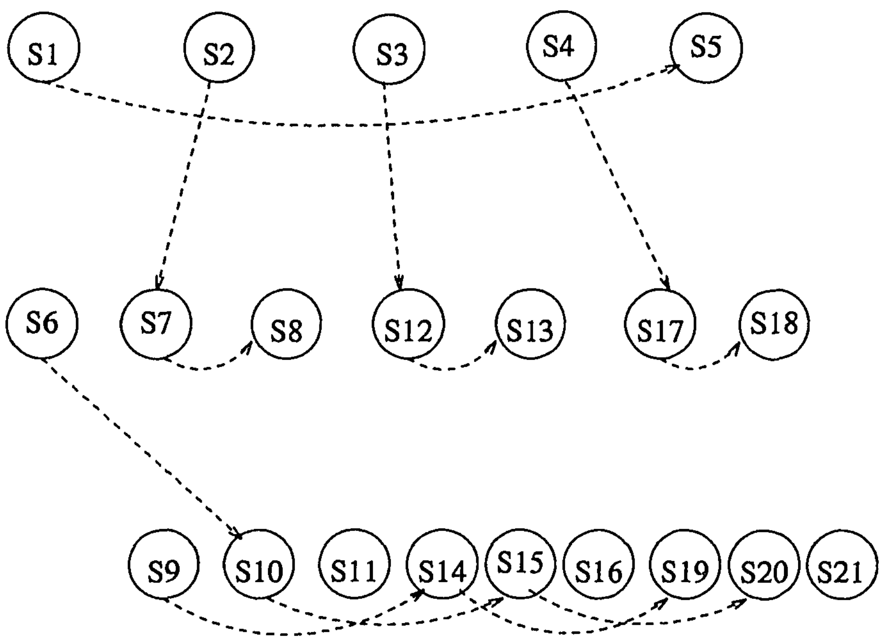


Figure 3.2 The DDG of procedure *VM_Simu*.

If two statements S_i and S_j use the same stores of data, and S_i executes before S_j, S_j is said to be *direct data-dependent* on S_i. This is denoted as $S_i \rightarrow_d S_j$. Data dependence describes contention for shared data. For example, in procedure *VM_Simu* (in Figure 3.1 (c)), statement $S1$ writes zero to variable *PECount* and statement $S5$ uses variable *PECount*. Thus, $S5$ must execute after $S1$, in order to get an initialized value for *PECount*. Therefore, $S5$ is direct data-dependent on $S1$. The *data dependence graph (DDG)* of a program is defined as (V, E) where V is a set of nodes called statement nodes (each statement node represents a statement in the program), and E is a set of edges where each edge represents a direct data-dependence relation between nodes. For example, the DDG of the procedure VM_Simu is shown in Figure 3.2.

A statement S_i is *direct control-dependent* on another statement S_j if the execution result of S_i determines whether S_j is executed or not. In an if-statement, for example, all the statements in both branches of the statement must wait for the

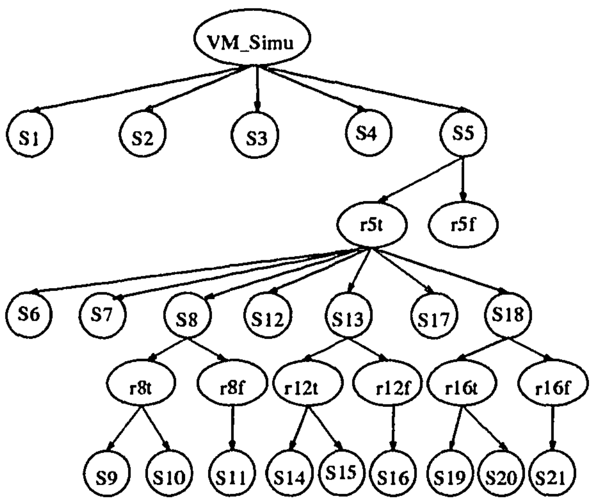


Figure 3.3 The CDG of the procedure *VM_Simu*.

evaluation of the condition to decide which branch has to be executed. Therefore, all the statements (in both branches) are control dependent on the condition evaluation statement. The *control dependence graph (CDG)* is defined as $G(V, R, E)$, where V is a set of statement nodes (each node represents a statement in the program), R is a set of nodes called *region nodes* (each node is used to group a set of statement nodes which have the same control dependence relations), and E is a set of edges where each edge represents a direct control-dependence relation between nodes (including region nodes). A *region node* is defined as a virtual node which has zero execution time. A region node is used to group all the nodes that have the same control dependence relation on the same node, by forcing all those nodes to depend on the region node. The root of the CDG is a special region node which indicates that all the statements of a program are control dependent on the activation of the program. For example, $S5$ of *VM_Simu* is a while-loop statement. The execution result of $S5$ determines whether the body of the loop is executed or not. Therefore, two region nodes ($r5t$ and $r5f$) are placed under node $S5$ to represent true and false results, respectively. All the statement nodes in the loop body are control dependence on region node $r5t$. Another purpose of using region nodes is that sibling region nodes are alternative execution paths of which only one is chosen during run-time. The sibling region nodes are called *mutually exclusive regions nodes.* For example, for each if-statement, two region nodes are created, one for all the statements in the true branch, another for all the statements in the false branch. These two region nodes are mutually exculsive, only statements under one of the two region nodes alive whenever the subprogram that contains the if-statement is called. The CDG of the procedure VM_Simu is shown in Figure 3.3.

The construction of PDGs in [19] relies on control flow graphs (CFGs), which capture required flow-of-control. The CFG of procedure *VM_Simu* is shown in Figure 3.4. For conditional statement, multiple possible paths are created. For example, statement $S8$ is an if-statement, two alternative paths are created to represent the two possible execution flow depending on the result of $S8$. For loop statement, beside the flow through the body of the loop, two additional flow paths are possible: skipping the loop body and going back to the loop control statement. For example, the result of the while-loop statement $S5$ determines whether to flow into the loop body or skip the loop body. After the execution of the last statement $S20$ or $S21$ of the loop body, control flows back to $S5$. In [19], a control dependence graph (CDG) which encodes control dependence only is constructed from a CFG. Then, the data dependence analysis is performed on the CDG. Finally, the PDG is constructed by adding data dependence edges to the CDG. By adding data dependence edges (dashed arrows) to the CDG, a PDG is constructed which describes both control and data dependence relations among the statements in a method. The PDG of procedure VM_Simu() is shown in Figure 3.5.

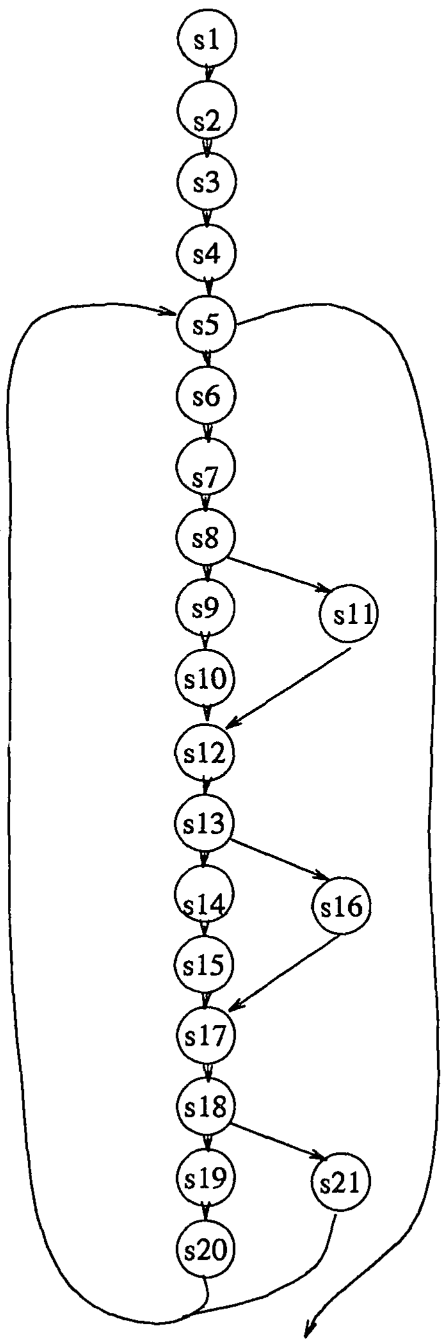


Figure 3.4 The CFG of procedure *VM_Simu*.

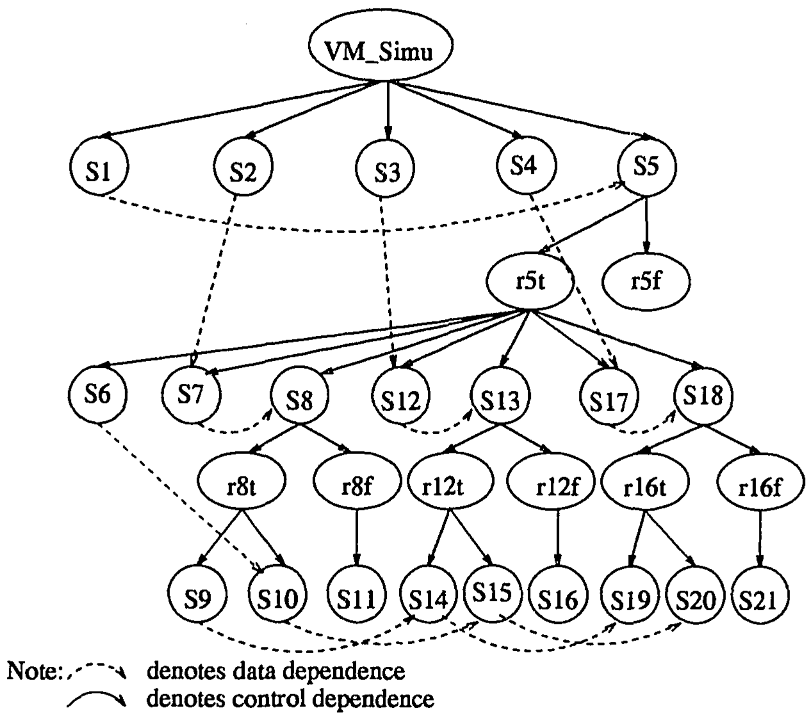


Figure 3.5 The PDG of the procedure *VM_Simu*.

3.2 Construction of Program Dependence Graphs

In this section, new algorithms for building DDGs, CDGs, PDGs, and SPGs are presented. In [19], more general programming (goto, recursive, etc.) model is considered. The algorithms presented in this thesis handles object-based structured programs. Therefore, they are more efficient in time and space. The overview of the construction process is shown in Figure 3.6. First, a *statement table (StaTab)* is constructed for each subprogram. The statement table of a subprogram is used to build the CFG, DDG, and CDG for a subprogram. The program dependence graph of a subprogram is created by combining the DDG and CDG of a subprogram. Finally, the data and control dependence relations are replaced by precedence relations.

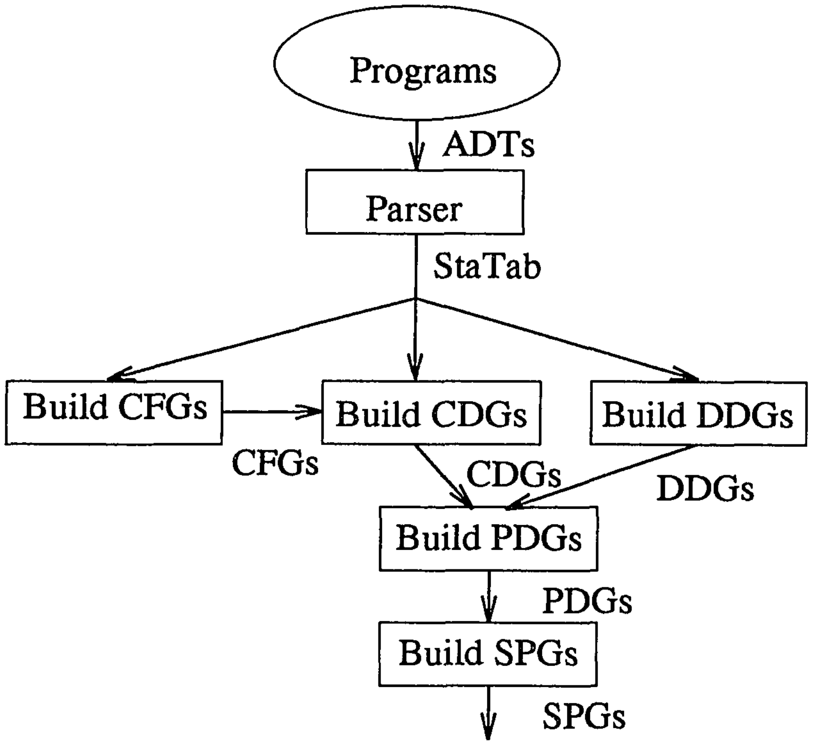


Figure 3.6 Overview of Dependence Graph Construction.

3.2.1 Statement Tables

There are many ways to construct program dependence graphs [19, 17, 4, 27]. For the programming language model used in this thesis, the author has devised a technique for constructing PDGs. First, statement tables are constructed. Each source statement has an entry in the statement table. Each entry consists of the following attributes:

1. *Statement Type* indicates the type of the statement (e.g. *method call*, *if-then-else*, *while loop*).

2. *Dependence Nesting Level* keeps track of the number of *region nodes* on the path from the root to the statement.

3. *Statement Address* is the line number in the source code.

4. *Used ADT Instances* is the set of ADT instances directly used by the statement.

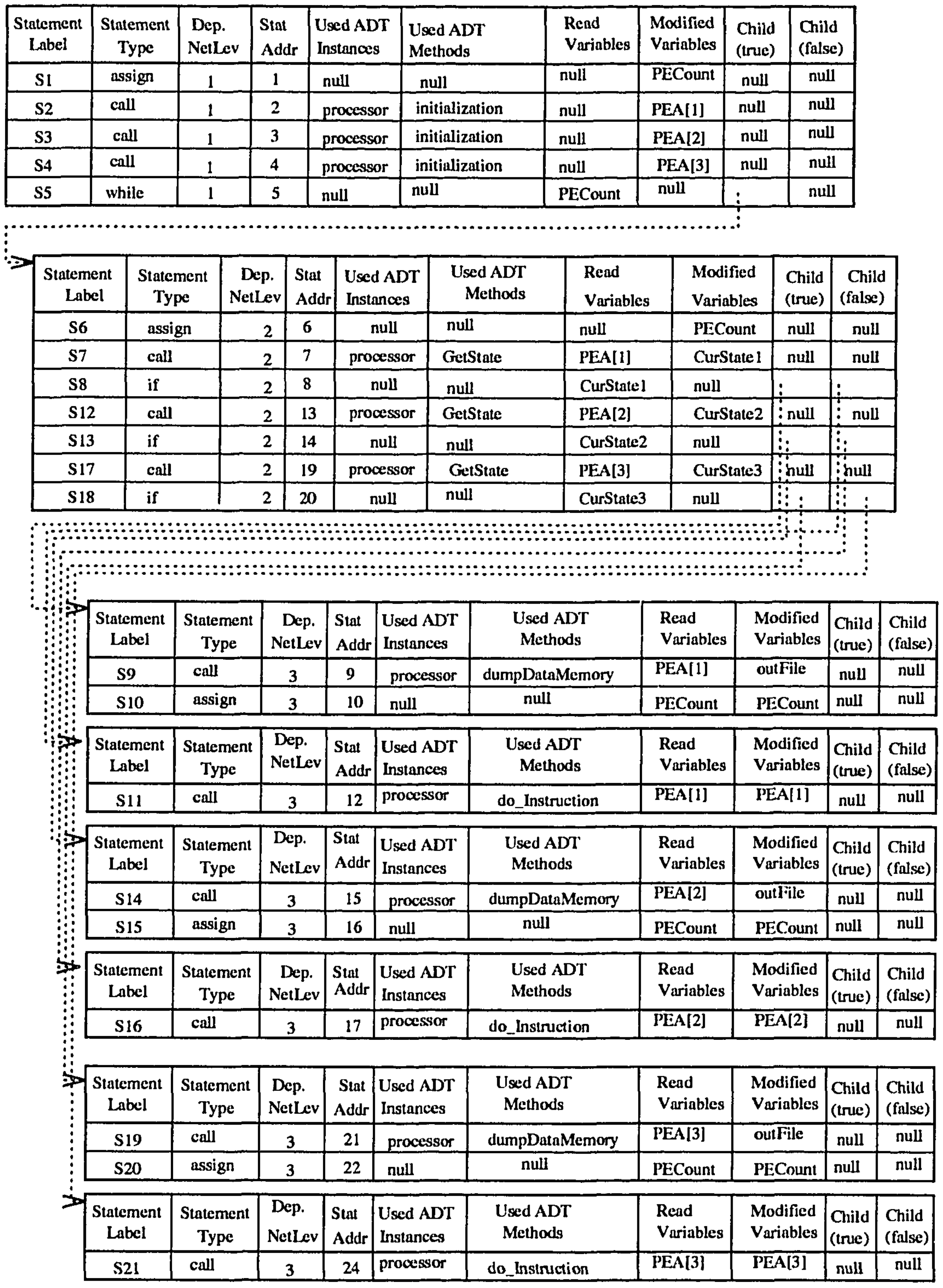
5. *Used ADT methods* is the set of ADT methods directly used by the statement.

6. *Read Variables* is the set of variables read (only) by the statement.

7. *Modified Variables* is the set of variables modified by the statement.

8. *Child* points to a statement table containing all dependents in another dependence level. If a statement has more than one group of statements depending on it, a *Child* field is created for each of the groups. This occurs when the statement is an *if-statement*, *case*, or *loop*, where there exist multiple execution paths. For all statements which do not create multiple branches, the child field is a *null* pointer.

For procedure *VM_Simu*, the statement table is shown in Figure 3.7. In the statement table, two child fields are created for each *while* or *if* statement.



Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S1	assign	1	1	null	null	null	PECount	null	null
S2	call	1	2	processor	initialization	null	PEA[1]	null	null
S3	call	1	3	processor	initialization	null	PEA[2]	null	null
S4	call	1	4	processor	initialization	null	PEA[3]	null	null
S5	while	1	5	null	null	PECount	null		null

Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S6	assign	2	6	null	null	null	PECount	null	null
S7	call	2	7	processor	GetState	PEA[1]	CurState1	null	null
S8	if	2	8	null	null	CurState1	null		
S12	call	2	13	processor	GetState	PEA[2]	CurState2	null	null
S13	if	2	14	null	null	CurState2	null		
S17	call	2	19	processor	GetState	PEA[3]	CurState3	null	null
S18	if	2	20	null	null	CurState3	null		

Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S9	call	3	9	processor	dumpDataMemory	PEA[1]	outFile	null	null
S10	assign	3	10	null	null	PECount	PECount	null	null

Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S11	call	3	12	processor	do_Instruction	PEA[1]	PEA[1]	null	null

Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S14	call	3	15	processor	dumpDataMemory	PEA[2]	outFile	null	null
S15	assign	3	16	null	null	PECount	PECount	null	null

Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S16	call	3	17	processor	do_Instruction	PEA[2]	PEA[2]	null	null

Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S19	call	3	21	processor	dumpDataMemory	PEA[3]	outFile	null	null
S20	assign	3	22	null	null	PECount	PECount	null	null

Statement Label	Statement Type	Dep. NetLev	Stat Addr	Used ADT Instances	Used ADT Methods	Read Variables	Modified Variables	Child (true)	Child (false)
S21	call	3	24	processor	do_Instruction	PEA[3]	PEA[3]	null	null

Figure 3.7 The Statement Table of Procedure *VM_Simu.*

```
BuildCDG(StaTab : StaTab_TYPE, entry : NODE_TYPE)
  var Q: QUEUE of node;
    x, y, z: NODE_TYPE;
  begin
    ENQUEUE(entry, Q):
    while not EMPTY(Q) do
      begin
        x := FRONT(Q);
        DEQUEUE(Q);
        for each none NULL ChildStaTab C of x in the StaTab do
        /* ChildStaTab is either x.LeftC or x.RightC */
          begin
            if (x.Type = "if") then
              begin
                y := getRegionNode; /* get a new region node */
                insert(x,y,CDG); /* insert an edge from x to y in the CDG */
              end
            else
              y:=x;
            for each entry N in C do
              begin
                z := getNode(N); /* get a new node with the label, N.label */
                insert(y,z,CDG);
                ENQUEUE(z,Q);
              end for
          end for
      end while
  end BuildCDG
```

Figure 3.8 Algorithm for building a CDG of a method.

3.2.2 Building Control Dependence Graphs

A CDG of a method can be directly constructed from the statement table of the method, since the statement table describes the control dependence relations among statements. A special region node called *entry* is added to the CDG to group all the statements in the top level of the statement table together. Also, for a statement which has two or more branches (like an if or a loop statement), a *region node* is added to the CDG for each branch, i.e, a *region node* is created for each statement table. Thus the start of a branch is indicated by the region node, and the region node becomes control dependent upon the statement that branches. All statements in each branch are control dependent upon the region node.

The algorithm for building a CDG from a statement table is shown in Figure 3.8. The three loops contribute the complexity of the algorithm which is $O(S^3)$ (S

is the number of statements in a method). The CDG of the procedure *VM_Simu* (shown in Figure 3.3) is constructed by supplying the algorithm with the statement table of Figure 3.7.

3.2.3 Building Data Dependence Graphs

Data dependence analysis is necessary for any form of automatic parallelism detection. Data dependence relations are used to determine if two operations, statements, or iterations of a loop can be executed in parallel. Data dependence graphs describe the data dependence relationship among statements. The DDG of a method can be constructed simply from the control flow graph of the method by examining the data dependence relation along the control flow of the method. The algorithm for building the DDG of a method is presented in Figure 3.9 and Figure 3.10. The DDG of the procedure *VM_Simu*, obtained by applying the algorithms, is shown in Figure 3.2. Searching for the successor of a statement (*successiveStatament*) travels the statement table - a binary tree. In the worst case, it takes $O(S)$ (if the tree is linear). Therefore, the complexity of the algorithm for building a DDG of a method is $O(S^2)$.

3.2.4 Building Program Dependence Graphs

A program dependence graph of a method can be built from the control dependence graph and the control flow graph of the method. This can be done by examining the data dependence relations on the control flow edges, and adding any identified data dependence relations into the control dependence graph. Then, a graph which describes both control and data dependence relations is constructed. The algorithm for building the PDG of a method from its DDG and CDG is shown in Figure 3.11. The algorithm calls *BuildCDG* for building the CDG of a method as the base of the PDG. Inside the only loop of the algorithm, it calls *SearchDD* ($O(S^2)$ to add data dependence into the PDG. Therefore, the complexity of the

```
SearchDD(tt : StatementType)
  PS : stack(StatementType)
  begin
    if (tt.rightc $\neq$ null) or (tt.leftc $\neq$ null) then
      Push tt.rightc & tt.leftc into stack PS;
    else
      begin
        st = successiveStatement(tt);
        if (st $\neq$ null) then stack.push(st, PS);
      end
    while not stack.empty(PS) do
      begin
        st = stack.pop(PS);
        if (st $\neq$ null) then
          begin
            if (st.Parameters $\cap$ tt.Parameters = $\emptyset$) then
              begin
                DDG(tt,st) = true;
                Remove (st.Parameters $\cap$ tt.Parameters) from tt;
                if no more parameters in tt remain to be checked then
                  while not stack.empty(PS) do st = stack.pop(PS);
                else
                  begin
                    st = successiveStatement(tt);
                    if (st $\neq$ null) then stack.push(st, PS);
                    else flag = true;
                  end
              end
            else
              if (st.rightc $\neq$ null) or (tt.leftc $\neq$ null) then
                Push st.rightc & st.leftc into stack PS;
              else
                begin
                  st = successiveStatement(tt);
                  if (st $\neq$ null) then stack.push(st, PS);
                end
          end
        if (flag = true) then
          while not stack.empty(PS) do st = stack.pop(PS);
      end while
  end SearchDD
```

Figure 3.9 Algorithm for searching for data dependence.

```
successiveStatement(st : StatementType) returns StatementType;
  begin
    if (st.rightc ≠ null) or (st.leftc ≠ null) then
      begin
        if (st.leftc ≠ null) then
          return (st.leftc);
        if (st.rightc ≠ null) then
            return (st.rightc);
      end
    else
      if (st.sibling ≠ null) then
        return (st.sibling);
      else
        begin
          while (st.parent ≠ null and st.parent.sibling = null) do
            st := st.parent;
          if (st.parent ≠ null and st.parent.sibling ≠ null) then
            return (st.parent.sibling);
          else
            return null;
        end
  end successiveStatement
```

Figure 3.10 Algorithm for searching for the successive statement.

```
BuildPDG(StaTab, m)
  begin
    BuildCDG(StaTab, m);
    copy CDG to PDG;
    for each node $n$ in method $m$ do
      $s$ = searchDD($n$);
      add $n \rightarrow_d s$ into PDG(m);
      if $n$ is not the ancestor of $s$ in CDG then
        if parent($s$) is a region node which is the ancestor of $n$ in CDG then
          remove the edge from parent($s$) to $s$ in PDG;
          add an edge from $n$ to $s$ in PDG;
        end if
    end for;
  end BuildPDG
```

Figure 3.11 Algorithm for building a PDG of a method.

```
TransformToSPG(PDG)
  begin
    SPG = copyPDG(PDG);
    for each node $n$ in SPG do
      for each node $s$ ($s \neq n$) in SPG do
        if $n \rightarrow_c s$ and $n \rightarrow_d s$ then
          begin
            remove $n \rightarrow_c s$ and $n \rightarrow_d s$;
            add $n \rightarrow s$;
          end;
        else if $n \rightarrow_c s$ then
          begin
            remove $n \rightarrow_c s$;
            add $n \rightarrow s$;
          end;
        else if $n \rightarrow_d s$ then
          begin
            remove $n \rightarrow_d s$;
            add $n \rightarrow s$;
          end;
      end for;
    end for;
  end TransformToSPG
```

Figure 3.12 Algorithm for transforming a PDG into a SPG.

algorithm for building a PDG of a method is $O(S^3)$. Figure 3.5 shows the PDG for procedure *VM_Simu*.

3.2.5 Building Statement Precedence Graphs

Since both control and data dependence relations force one statement to wait for the completion of another, *precedence* is used to refer to either control dependence or data dependence relations. A precedence relation that requires the execution of statement s_i to precede the execution of statement s_j is denoted as $s_i \rightarrow s_j$. The PDG is transformed into a *statement precedence graph (SPG)* in which a solid arrow is used to describe a precedence relation. Thus, in an SPG, control dependence and data dependence are not distinguished. The SPG of the procedure *VM_Simu()* is shown in Figure 3.13. The algorithm for transforming a PDG of a method into a SPG of a method is presented in Figure 3.12 ($O(S^2)$).

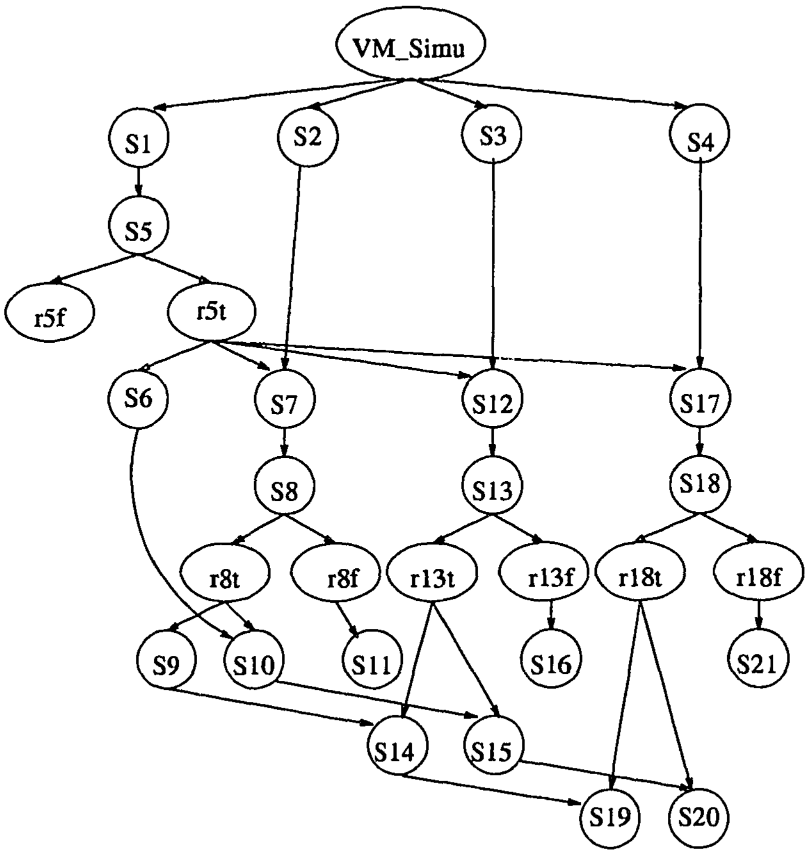


Figure 3.13 The SPG of the procedure *VM_Simu*.

3.3 Code Dependence Graphs

A PDG of a program describes control and data dependence relations among the statements in the program. Thus, the PDG indicates that any two statements can run concurrently if they have neither data nor control dependence relations. Note the above conclusion may not hold if the two statements call the same method. One call must wait for the other to complete since only one copy of code of the method is available. For example, statements s_2, s_3, and s_4 in Figure 3.13 call the same method (*Processor.initialization*); they cannot run concurrently even though they have neither direct nor transitive precedence relations (see the SPG of 3.13), since the code of a method cannot be used concurrently (though it may be reentrant) since only one PE contains a copy of the code. If multiple method calls try to execute the same method, only one of them is granted access at any instant; the others must wait since a CPU only executes one instruction at a time

To model this kind of contention, the author has coined the term *code dependence* relation, which is defined as follow. If two calls S_i and S_j require the same method m exported by an ADT instance f, S_i and S_j are said to have a *code dependence* relation. Code dependence is symmetric, and is therefore denoted as $S_i \overset{f.m}{\leftrightarrow} S_j$. Code dependence describes access contention among multiple clients for a method. Dependence $S_i \overset{f.m}{\leftrightarrow} S_j$ indicates that either statement S_j waits for the completion of statement S_i, or else S_i waits for S_j. It is important to describe method contention since ADT instances are often stateless. Data and data consistency are maintained by clients. Therefore, replication of methods exported by stateless ADT instances needs no consistency and synchronization control, and is potentially a large source of concurrency.

By adding code dependence relations into an SPG, a new graph describing precedence relations (control and/or data) and code dependence is obtained. This new graph is called the *general dependence graph* (GDG). There are three kinds

```
SearchMD(tt : StatementType)
  PS : stack(StatementType)
  begin
    if (tt.rightc ≠ null) or (tt.leftc ≠ null) then
      Push tt.rightc & tt.leftc into stack PS;
    else
      begin
        st = successiveStatement(tt);
        if (st ≠ null) then stack.push(st, PS);
      end
    while not stack.empty(PS) do
      begin
        st = stack.pop(PS);
        if (st ≠ null) then
          begin
            if (st.usedMethods ∩ tt.usedMethods = ∅) then
              begin
                GDG(tt,st) = M;
                Remove (st.usedMethods ∩ tt.usedMethods) from tt;
                if no more used method in tt remain to be checked then
                  while not stack.empty(PS) do st = stack.pop(PS);
                else
                  begin
                    st = successiveStatement(tt);
                    if (st ≠ null) then stack.push(st, PS);
                    else flag = true;
                  end
              end
            else
              if (st.rightc ≠ null) or (tt.leftc ≠ null) then
                Push st.rightc & st.leftc into stack PS;
              else
                begin
                  st = successiveStatement(tt);
                  if (st ≠ null) then stack.push(st, PS);
                end
          end
        if (flag = true) then
          while not stack.empty(PS) do st = stack.pop(PS);
      end while
  end SearchMD
```

Figure 3.14 Algorithm for searching for method dependence).

```
BuildGDG(StaTab, m)
  begin
    PDG = BuildPDG(StaTab, m);
    SPG = TrasnformToSPG(PDG);
    copy PDG to GDG;
    for each node $n$ in method $m$ do
      $s$ = searchMD($n$);
      add $n \rightarrow_m s$ into GDG(m);
    end for;
  end BuildGDG
```

Figure 3.15 Algorithm for building a GDG of a method.

of edges and three kinds of nodes in the GDG of a method. Directed solid edges represent precedence relations, and undirected dashed edges represent code dependence relations. The labels on undirected dashed edges are method names (which are used by source and destination nodes). Nodes within solid circles represent local statements (i.e., they are not method calls). Nodes within dashed circles are call nodes, which represent call statements. Nodes within solid ovals are region nodes. For the procedure *VM_Simu*, the GDG(*VM_Simu*) is shown in Figure 3.16. The directed solid edge (s_1, s_5) represents a precedence relation between statements s_1 and s_5. The undirected dashed edge (s_2, s_3) represents a code dependence relation between statements s_2 and s_3. Node $S1$ (within a solid circle) represents a local statement. Node $S2$ (within a dashed circle) represents a call statement. Node $r5t$ (within a solid oval) represents a region node.

The algorithm for building a GDG of a method is presented in Figure 3.15. It calls the function for constructing the PDG and uses the PDG as the base of the GDG. Inside the only loop of the algorithm, it calls *SearchMD* ($O(S^2)$ to add code dependence into the GDG. Therefore, the complexity of the algorithm for building a GDG of a method is $O(S^3)$.

A subgraph of a method's GDG consisting of nodes that call the same method is called a *method dependence subgraph (MDG)* of the method. For example, in Figure 3.16, nodes $s1$, $s2$, and $s3$ of procedure *VM_Simu* call method $m1$, therefore, these three nodes and the edges between them form $MDG(m1, VM\_Simu)$ as shown in Figure 3.17 (a). In other words, they constitute the set of statements in procedure *VM_Simu* that used method $m1$. Nodes s_7, s_{12}, and s_{17} call method $m2$, thus, they form MDG($m2, VM\_Simu$) as shown in Figure 3.17 (b). MDG($m3, VM\_Simu$) contains statements s_{11}, s_{16}, and s_{21} and is shown in Figure 3.17 (c). MDG($m4, VM\_Simu$) contains statements s_9, s_{14}, and s_{19} and is shown in Figure 3.17 (d).

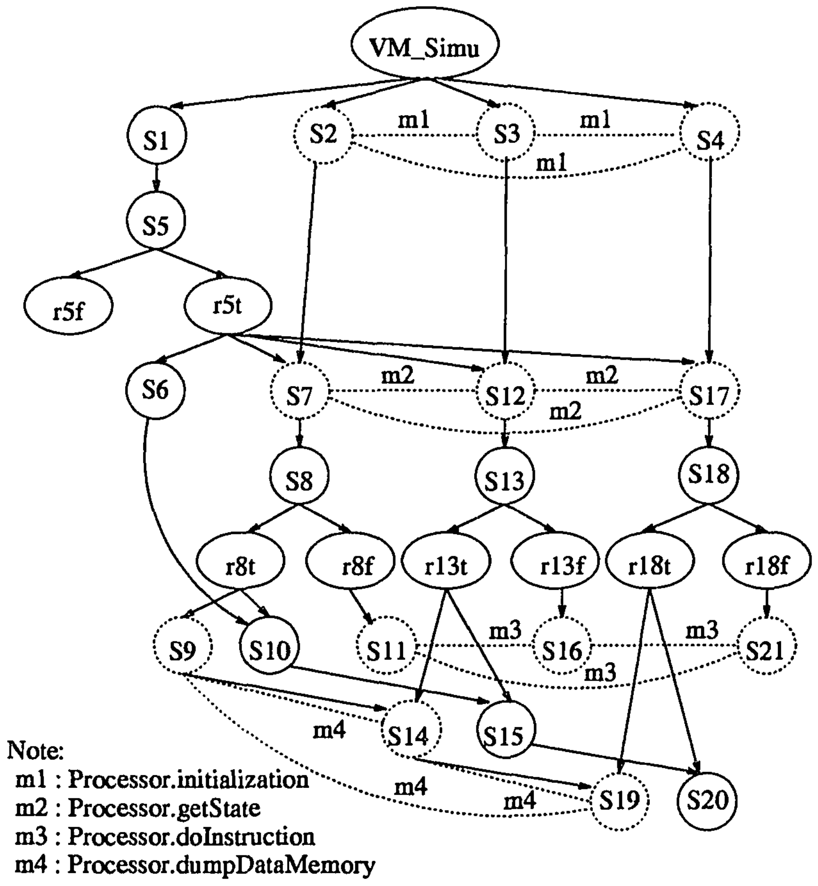


Figure 3.16 The GDG of procedure *VM_Simu*.

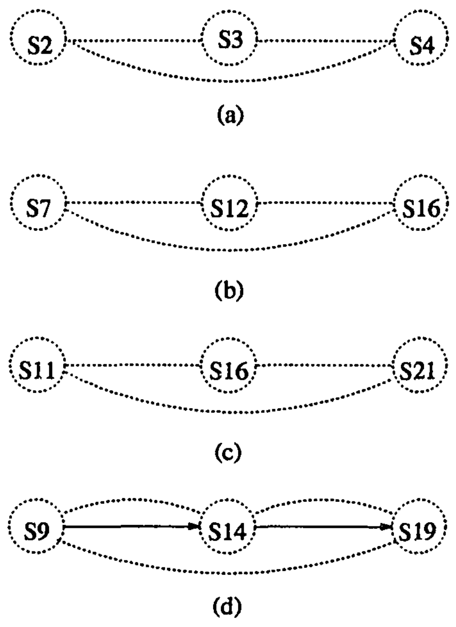


Figure 3.17 (a) MDG(ml, VM_Simu). (b) MDG(m2, VM_Simu). (c) MDG (m3, VM_Simu). (d) MDG(m4, VM_Simu).

Since code dependence is described with an undirected edge, all the nodes in the MDG form a complete graph. The number of MDGs in a method's GDG is equal to the number of methods used by the method. For example, *VM_Simu* has four used methods, thus it has four MDGs.

Since the execution model used in this work (see Section 2.2) processes the statements of a method unit in sequence (i.e., each processing element is assumed to be a van Neumann processor), statements have relative order. Thus, all code dependence relations are changed from undirected edges to directed edges by incorporating the relative positions of statements. For example, the simplified MDGs of procedure *VM_Simu* are shown in Figure 3.18.

3.4 Ineffective Code Dependence

GDGs may contain some method dependence relations that do not reflect contention that can actually occur due to other precedence relations. For example, if two nodes

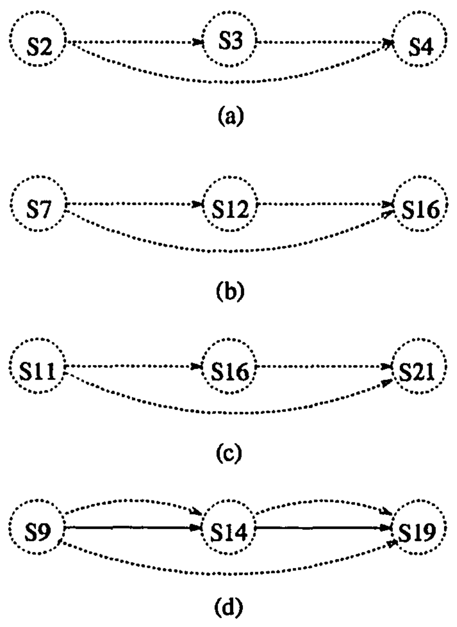


Figure 3.18 (a) MDG(ml, VM_Simu). (b) MDG(m2, VM_Simu). (c) MDG (m3, VM_Simu). (d) MDG(m4, VM_Simu) after incorporating relative statement positions.

have both precedence and code dependence relations, the precedence relation forces the two statements to run sequentially, thus the code dependence is *ineffective*, i.e., it falsely indicates contention for the code of the method. Such code dependence relations are called *ineffective code dependence relations.* This section presents techniques to identify and remove these from GDGs.

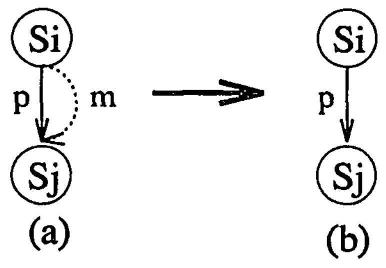


Figure 3.19 A precedence relation makes a code dependence ineffective.

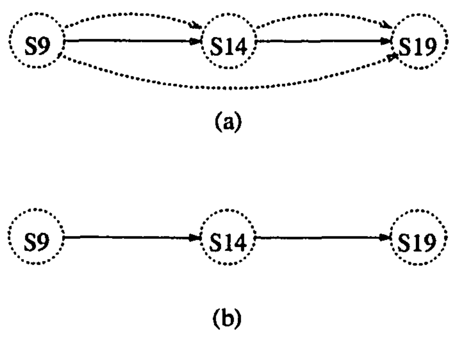


Figure 3.20 (a) *MDG(m4, VM_Simu)* before applying Theorem 1. (b) *MDG(m4, VM_Simu)* after applying Theorem 1.

The following theorems show how a precedence relation affects code dependence relations. Assume two statements s_i and s_j have a precedence relation and that they both call the same method m. Since the precedence relation forces the two statements to run sequentially, the code dependence falsely indicates contention, and it can be removed. This fact is formally stated as:

Theorem 1 *If* $s_i \rightarrow s_j \wedge s_i \xrightarrow{m} s_j$, *then* $s_i \xrightarrow{m} s_j$ *is ineffective.*

Proof: when two statements s_i and s_j have both precedence and code dependence relations (as shown in Figure 3.19), the code dependence does not indicate true contention, since the precedence relation forces the two statements to run sequentially. Therefore, the code dependence between s_i and s_j is ineffective. It can be removed (as shown in Figure 3.19 (b)). $\square$

Another example is shown in Figure 3.20 (a), where the precedence relation $s9 \rightarrow s14$ makes $s9 \rightarrow_m s14$ ineffective, and the precedence relation $s14 \rightarrow s19$ makes $s14 \rightarrow_m s19$ ineffective. The transitive precedence relation between $s9$ and $s19$ also makes $s9 \rightarrow_m s19$ ineffective.

Assume that there is a precedence relation $s_i \rightarrow s_j$, and that node s_k has code dependence relations with statements s_i and s_j in $MDG(m, n)$ (i.e., $s_i \overset{m}{\leftrightarrow} s_k$ and $s_k \overset{m}{\leftrightarrow} s_j$). Furthermore, assume that there is no precedence relation between s_k and

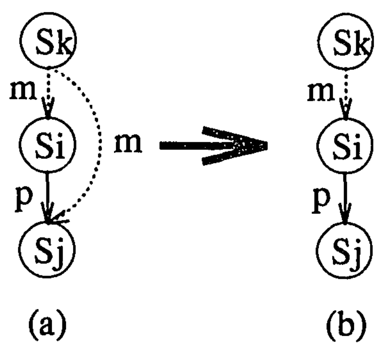


Figure 3.21 Removing an ineffective code dependence between s_k and s_j.

s_i, nor between s_k and s_j. Then there are only three sequential execution orders of s_i, s_j, and s_k which preserve the semantics of the program. The three possible orders are $(s_k,\ s_i,\ s_j)$, $(s_i,\ s_k,\ s_j)$, and $(s_i,\ s_j,\ s_k)$. Since s_i and s_j can not run concurrently (due to the precedence relation between them), at most two of the three statements can run concurrently. Therefore, two clones of m are sufficient to resolve contention. The following three theorems describe how the precedence relations affect the code dependence relations in each of the three cases.

Theorem 2 *If* $s_i \dashrightarrow s_j \wedge s_k \xrightarrow{f.m} s_i \wedge s_k \xrightarrow{f.m} s_j$, *then* $s_k \xrightarrow{f.m} s_j$ *is ineffective.*

Proof: Statement s_j does not contend with statement s_k for method m since statement s_j must wait for the completion of statement s_i (because of the precedence relation $s_i \dashrightarrow s_j$). Therefore, $s_k \xrightarrow{f} s_j$ does not describe actual code contention between statements s_k and s_j. Thus, $s_k \xrightarrow{f} s_j$ is ineffective, and can be removed as shown in Figure 3.21 (b). □

Theorem 3 *If* $s_i \dashrightarrow s_j \wedge s_i \xrightarrow{f.m} s_k \wedge s_k \xrightarrow{f.m} s_j$, *then* $s_k \xrightarrow{f.m} s_j$ *is ineffective.*

Proof: Statement s_j does not contend for method m with statement s_k since statement s_j must wait for the completion of statement s_i due to the precedence

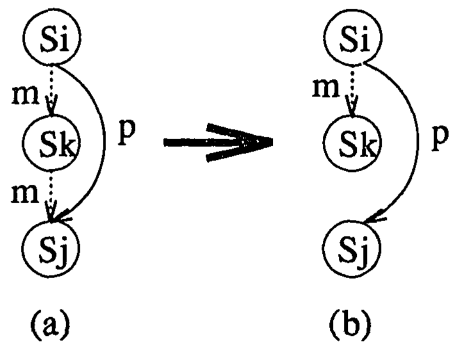


Figure 3.22 Removing ineffective code dependence between s_k and s_j.

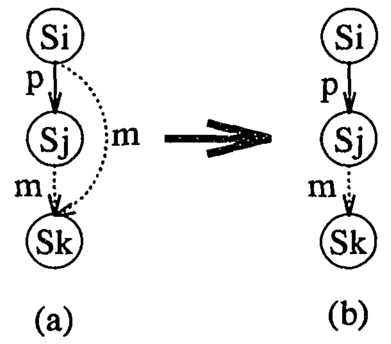


Figure 3.23 Removing ineffective code dependence between s_i and s_k.

relation $s_i \rightarrow s_j$. Therefore, $s_k \xrightarrow{f} s_j$ does not describe actual code contention between statements s_k and s_j. Thus, $s_k \xrightarrow{f.m} s_j$ is ineffective, and can be removed as shown in Figure 3.22 (b). □

Theorem 4 *If $s_i \rightarrow s_j \wedge s_i \xrightarrow{f.m} s_k \wedge s_j \xrightarrow{f.m} s_k$, then $s_i \xrightarrow{f.m} s_k$ is ineffective.*

Proof: Statement s_j is blocked until the completion of statement s_i due to the precedence relation $s_i \rightarrow s_j$. Statement s_k is not processed until after statement s_j is processed in a Von Meuman processor. Therefore, $s_i \xrightarrow{f.m} s_k$ does not describe actual code contention between statements s_i and s_k. Thus, $s_i \xrightarrow{f.m} s_k$ is ineffective, and can be removed as shown in Figure 3.23 (b). □

```
RemoveIneffectiveMD ( MDG($f.m$, $g.op$) )
 begin
  for each $s_i \rightarrow s_j$ in MDG($f.m$, $g.op$)
   if $s_i \xrightarrow{f.m} s_j$ then
    Remove $s_i \xrightarrow{f.m} s_j$ /* by Theorem 1 */
    for each node $s_k$ such that $s_k \xrightarrow{f.m} s_i$ and $s_k \xrightarrow{f.m} s_j$ in MDG($f.m, g.op$)
     case $k < i < j$ or $i < k < j$ :
      Remove $s_k \xrightarrow{f.m} s_j$ /* by Theorem 2 and 3 */
     case $i < j < k$ :
      Remove $s_k \xrightarrow{f.m} s_i$ /* by Theorem 4 */
    end for
   end if
  end for
 end
```

Figure 3.24 The algorithm for removing ineffective code dependences.

The four theorems above discuss the effects of precedence relations on code dependence in an MDG of a method. Based on those theorems, the algorithm of Figure 3.24 removes all ineffective code dependences. The algorithm scans for precedence relations in a MDG, it removes code dependence relation between nodes that have a precedence relation (applying Theorem 1). Then, the algorithm checks every other node to see if the node has code dependence relations with both of the two nodes that have the precedence relation, and if so, then applies one of theorems 2, 3, or 4. The time complexity of the algorithm is $O(E \cdot S) = O(S^3)$, where E is the number of edges in the MDG which is at most the number of nodes in the MDG.

For the virtual machine simulation, $GDG(VM\_Simu)$ after removing ineffective code dependence relations is shown in Figure 3.25.

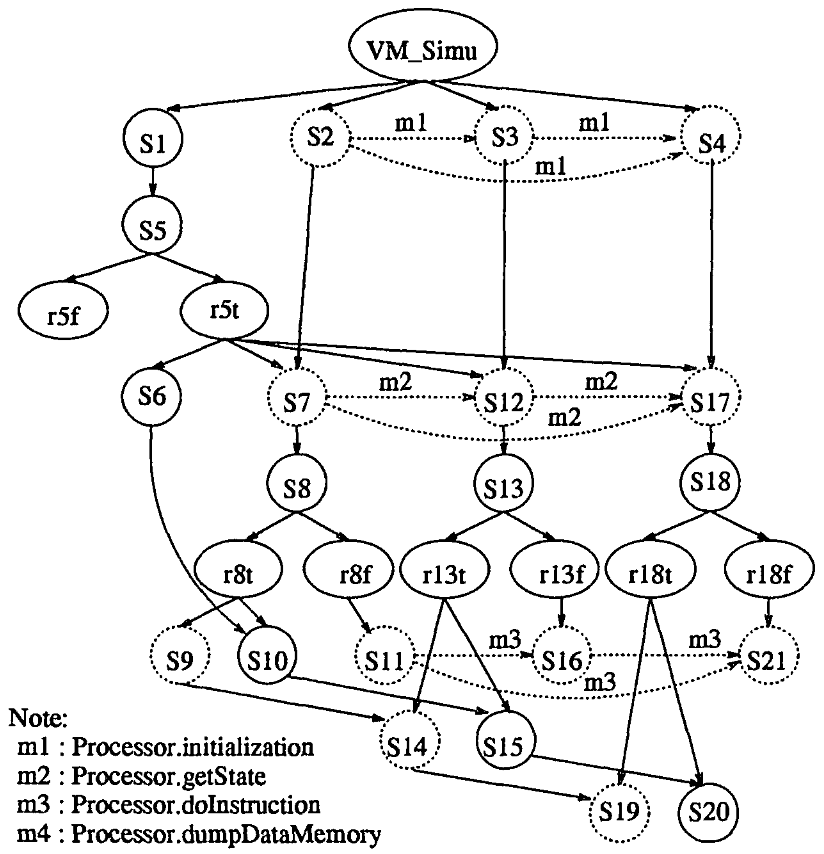


Figure 3.25 *GDG(VM_Simu)* after removing ineffective code dependence relations.

CHAPTER 4

CLONING ANALYSIS

In this chapter, a set of analysis techniques are presented for determining the clone requirement of each method to resolve all possible code contention. First, the method call graph is introduced. In Section 4.3, it is shown how the direct clone requirement of each method is determined by using the GDG of the method. To calculate cloning requirements for transitively used methods, Section 4.4 discusses inter-method dependence and cloning analyses. Techniques for handling loops are discussed in Section 4.5. Optimization techniques for exploitation of concurrency are discussed in Section 4.6. Techniques to aggregate clone requirements of methods to determine clone requirements of ADT instances are discussed in Section 4.7.

4.1 Overview of Dependence and Cloning Analysis Approach

In order to exploit concurrency automatically, the dependence relations among statements, among methods, and among instances are analyzed. Techniques are developed to determine the upper bound on the number of clones of methods and instances that may be used concurrently, - the *clone requirement (CR)* of the method. Intuitively, CR is the minimum number of clones needed to resolve all the possible contention.

Figure 4.1 shows the overview of the dependence and cloning analysis approach. The parser builds the *symbol table (SymTab)* and the *statement table (StaTab)* for each method. These two tables are used to construct CFG, CDG, and DDG for each method. The PDG of each method is constructed by combining the CFG, CDG, and DDG of the method. Since both data and control dependence relations force a sequential execution of statements, precedence relations are used to describe the sequencing of the statements. A SPG of a method is constructed from the PDG of the method to describe the precedence relations among statements in the method. By adding code dependence relations into the SPG of a method, a GDG of the method is constructed. With the GDG of a method, a MDG is constructed for each of its used methods. If a method used by more than one client method, the dependence relations among the client methods need to be analyzed. The dependence relations among methods are described by a method call graph. An *inter-method code dependence graph (IMDG)* of a method is constructed to for describing the dependence relations for a shared method. For each method, the number of clones of its directly used methods needed to resolve all contention is called *direct clone requirement (DCR).* DCR of each method is determined by finding the maximum possible contention in the GDG of a method. The maximum possible contention is determined by finding the node that has the maximum out going code dependence edges in the MDG of the method. To calculate the total number of *clone requirements (CR)* of beth directly and transitively used methods, by following the reverse topological order of the method call graph, the DCRs and CRs at lower level of MCG are propagated to CRs at higher level of the MCG.

4.2 Method Call Graph

For cloning analysis, it is not only necessary to model statement-level relationships, but also method- and instance-level relationships. If a method m exported by an ADT instance f is called by method n exported by an ADT instance g ($g \neq f$), method m is said to be *directly used* by method n. For example, in procedure *VM_Simu*, statements $s2$, $s3$, and $s4$ call method *Processor.initialization*. Therefore, *Processor.initialization* is directly used by *VM_Simu*. If n is directly used by method p exported by an ADT instance h ($h \neq g \neq f$), then m is said to be *transitively (or indirectly) used* by p (recall that n calls m). All the

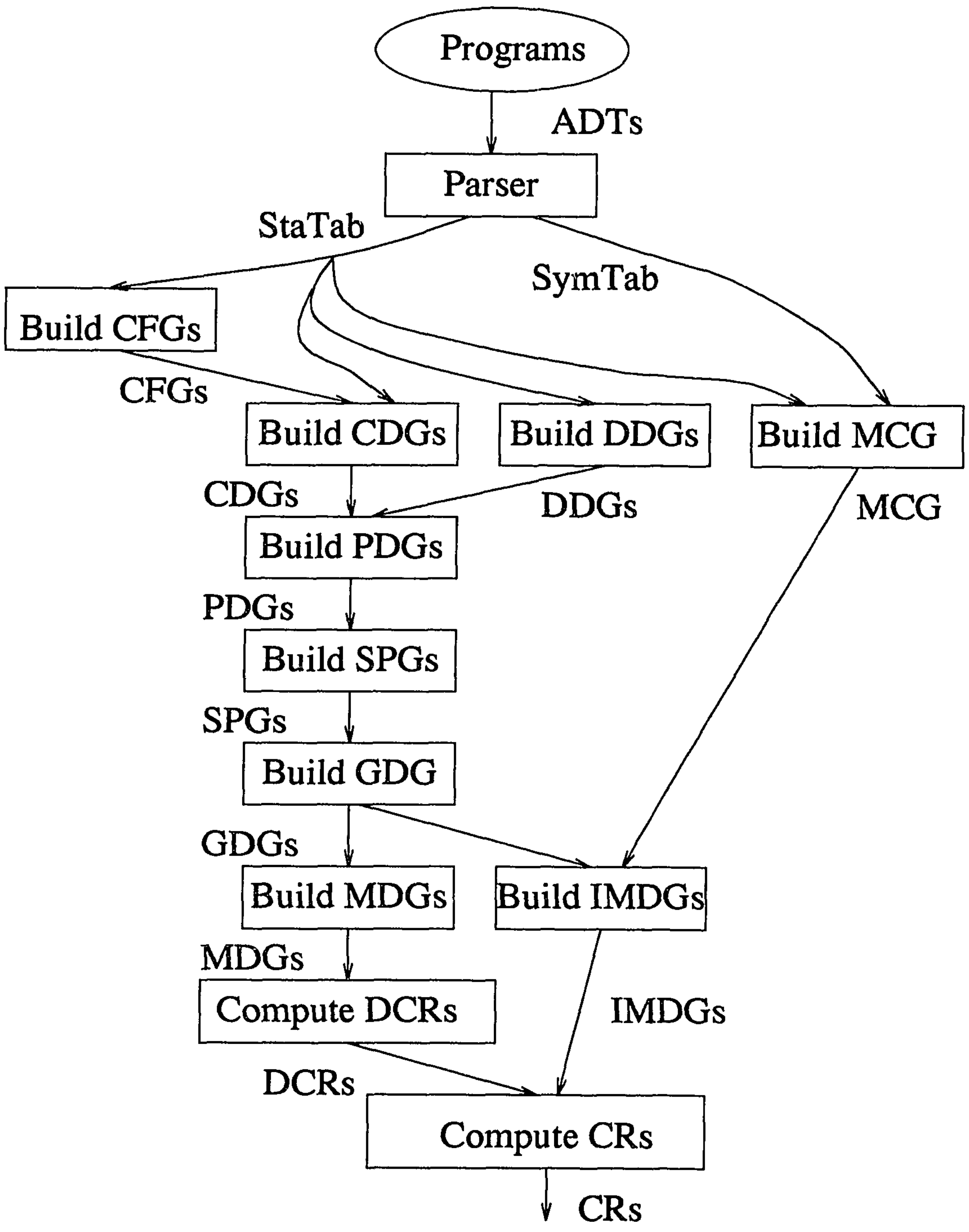


Figure 4.1 The flowchart of dependence and cloning analysis.

```
buildMCG(app)
 begin
   for each method $m$ of the app do
     create a node $N(m)$ for $m$ in the MCG if it does not exist;
     for each statement $s$ in $m$ do
       if $s$ calls method $p(p \neq m)$ then
         create a node $N(p)$ for p in the MCG if it does not exist;
         add an edge $m \rightarrow p$ in the MCG if it does not exist;
       end if;
     end;
   end for;
 end;
```

Figure 4.2 Algorithm for constructing a MCG.

methods transitively used by m are also transitively used by p. For example, method *InstrMem.initialize* is directly used by method *Processor.initialization*, therefore, method *InstrMem.initialize* is transitively used by procedure *VM_Simu*.

The direct use relation among methods is described with a directed graph called *method call graph* (MCG). In an MCG, nodes represent methods, and edges represent the direct use relation between methods. Since all methods are declared before being used, the MCG of a method is an acyclic graph. If a method is used by only one method, the former is called a *private method* of the latter. If a method is used by more than one method, it is called a *shared method* of the latter. The leaves of the MCG (have no used method) are called *primitive methods.* The internal nodes of the MCG (have at least one used method) are called *synthesized methods.* The algorithm for constructing a MCG of an application is presented in Figure 4.2. It scans the statement table of each method and constructs the call graph. The complexity of the algorithm is $O(M \cdot S)$ (M is the number of methods in an application, and S is the maximum number of statements in a method).

The MCG of the virtual machine simulation is shown in Figure 4.3. Method *InstrMem.initialize* is directly used by method *Processor.initialization.* Method *InstrMem.initialize* is a private method of method *Processor.initialization.* Method *DataMem.fetch* is used by both *Processor.doInstruction* and *Process-*

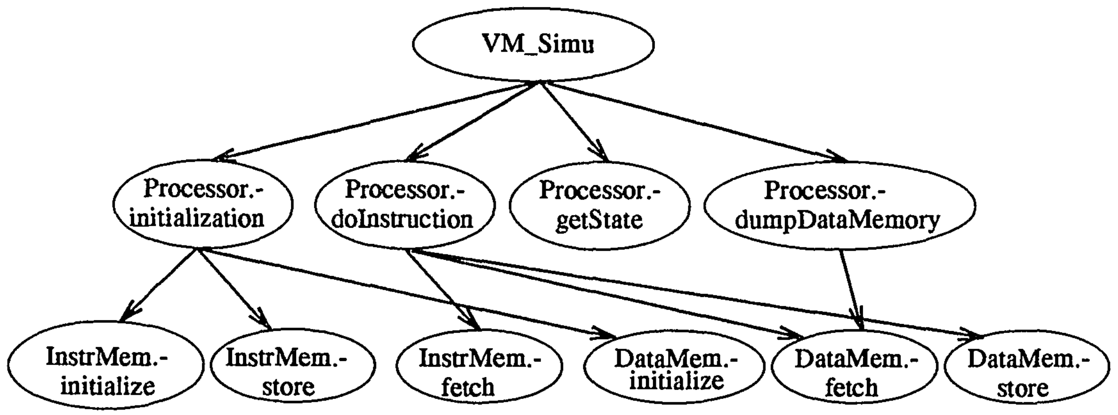


Figure 4.3 The method call graph of the virtual machine simulation.

or.dumpDataMemory, so it is a shared method. Method *InstrMem.initialize* is transitively used by procedure *VM_Simu*.

A method which does not call any other methods is called a *primitive method.* A method which calls other methods is called a *synthesized method.* When a call statement s of a method m calls a method n, if method n is not a primitive method, it further calls other methods; each of the called methods continues making calls to other methods, until a primitive method is called; the primitive method stops the calling sequence and starts a returning sequence by following the reverse order of the calling sequence. For example, statement $s2$ in *VM_Simu* calls method *Processor.initialization*, and statements in *Processor.initialization* further call method *InstrMem.initialize*, method *InstrMem.store*, and method *DataMem.initialize.* Since the latter three methods are all primitive methods, calling sequences are stopped and return sequences begin with them. Such a calling sequence is denoted as: $< s_i, s_{i+1}, \cdots, s_j >$. The first call on a calling sequence c is called the *front* of the calling sequence denoted as $front(c)$, and the rest of calls are called *tail* of the calling sequence denoted as $tail(c)$. One statement, may originate more than one calling sequence, due to the fact that a method can call more than one method. The set of all calling sequences starting at statement s is called the *calling sequence set*

(CSS) of s, and is denoted as $CSS(s)$. The call statement s is a statement which is the starting call of all the calling sequences in the $CSS(s)$. The method that s calls is called All the methods called by the calls on the calling sequences in $CSS(s)$ are called *used methods* of statement s (denoted $UM(s)$). Obviously, the method called by the front call of a calling sequence is directly used method by s; and the methods called by the tail calls of the calling sequence are transitively used methods by s. For example, starting at statement $s2$ in VM_Simu, there are three calling sequences, therefore,

$$\begin{aligned} CS(s_2) = \{ & < Processor.initialization.InstrMem.initialize >, \\ & < Processor.initialization.InstrMem.store >, \\ & < Processor.initialization.DataMem.initialize > \} \end{aligned}$$

The set of all directly used methods by a method m is called the *directly used method set (DUMS)* of m defined as:

$$DUMS(m) = \bigcup_{\forall s \in S(m), \forall i \in CSS(s)} methodcalledbyfront(i)$$

where $S(m)$ denotes the statement set of method m.

$DUMS$ can be directly constructed from the statement table of m. For example, methods *initialization*, *doInstruction*, *getState*, and *dumpDataMemory* exported by ADT instance *Processor* are directly called by the statements in procedure VM_Simu as shown in Figure 4.3. Therefore,

$$\begin{aligned} DUMS(VM\_Simu) = \{ & Processor.initialization, Processor.doInstruction, \\ & Processor.getStateProcessor.dumpDataMemory \} \end{aligned}$$

The union of all transitively used methods by a method m is called the *transitively used method set* of method m defined as:

$$TUMS(m) = \bigcup_{\forall s \in S(m), \forall i \in CSS(s)} methodscalledbytail(i)$$

For example, the transitively used methods are *initialize*, *store*, *fetch* exported by ADT instances *InstrMem* and *DataMem*, respectively. Therefore,

$$\begin{aligned} TUMS(VM\_Simu) = \{ & InstrMem.initialize, InstrMem.store \\ & InstrMem.fetch, DataMem.initialize \\ & DataMem.fetch, DataMem.store \ \} \end{aligned}$$

The union of all (both direct and transitive) used methods of the statements in a method m is called the the *used method set* of method m denoted as $UMS(m)$.

$$UMS(m) = \bigcup_{\forall s \in S(m)} UM(s) = DUMS(m) \cup TUMS(m)$$

where $S(m)$ denotes the statement set of method m. For example, the procedure *VM_Simu* in Figure 3.1 has 12 method calls. Four methods (*initialization*, *getState*, *doInstruction*, and *dumpDataMemory*) exported by instance *Processor* are called. Method *initialization* further calls methods *initialize* and *store* exported by instance *InstrMem*, and method *initialize* exported by instance *DataMem*. Therefore,

$$\begin{aligned} UMS(VM\_Simu) = \{ & Processor.initialization, Processor.getState \\ & Processor.doInstruction, Processor.dumpDataMemory \\ & InstrMem.initialize, InstrMem.store \\ & InstrMem.fetch, DataMem.initialize \\ & DataMem.fetch, DataMem.store \ \} \end{aligned}$$

Note that $DUMS(m) \subseteq UMS(m)$ and $TUMS(m) \subseteq UMS(m)$. Also, it is not always true that $DUMS(m) \bigcap TUMS(m) = \phi$ since a method may be used both directly and transitively.

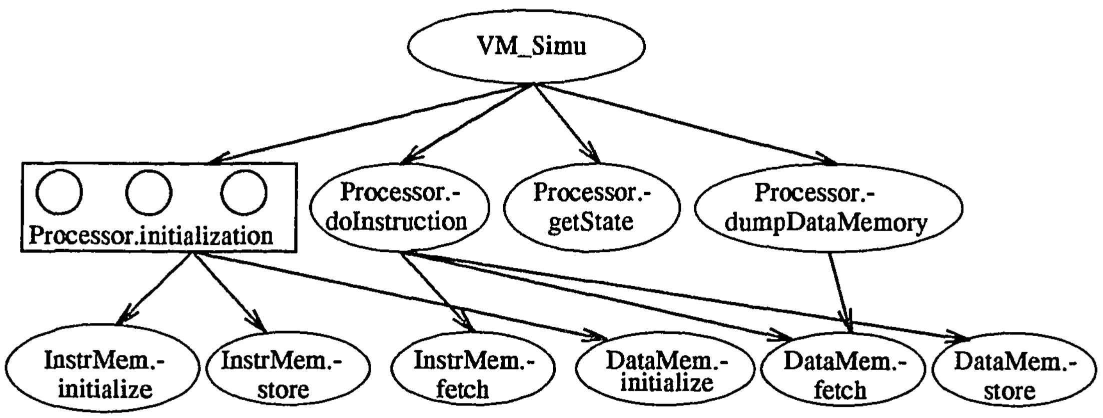


Figure 4.4 Three clones of method *Processor.initialization* are made.

4.3 Direct Clone Requirements

Code dependence between two statements indicates possible contention for the method they call. By cloning the method, the contention is resolved, and the two calls can be served concurrently by the two clones. For example, in Figure 3.18 (a), statements $s2$, $s3$, and $s4$ have no precedence relations among them, but the code dependence relations prevent them from executing concurrently. Clones of method *Processor.initialization* can resolve the contention among $s2$, $s3$, and $s4$ as shown in Figure 4.4. However, an infinite number of clones of method *Processor.initialization* cannot be utilized effectively. An upper bound on the number of clones that can be used concurrently exists. This section shows how to compute this upper bound.

Assume a method m exported by an ADT instance f is directly used by a method n exported by another ADT instance g ($g \neq f$). To resolve all possible contention for method m among the calls in method n, the minimum number of clones of m needed is called the *direct clone requirement (DCR)* of method m by method n. This is denoted as DCR($f.m$, $g.n$).

Recall that the code dependence edges in the MDG of a method describe the contention for a used method. The node with the largest out degree (OD) in the method indicates the maximum contention. To resolve all possible contention for the method, the number of clones must equal the largest out degree. Therefore, the DCR of a used method can be determined by finding the largest out degree of code dependence edges among the nodes of the MDG of the used method. This is stated formally as:

Let $OD(s, g.n)$ denote the out degree of statement s in $MDG(f.m, g.n)$, then

$$OD(MDG(f.m, g.n)) = MAX_{s \in MDG(f.m,g.n)}(OD(s, g.n))$$

For example, in $MDG(Proc.init, VM\_Simu)$ (as shown in Figure 3.18 (a)), $OD(s2, VM\_Simu) = 2$, $OD(s3, VM\_Simu) = 1$, and $OD(s4, VM\_Simu) = 1$. Therefore, $OD(MDG(Proc.init, VM\_Simu)) = 2$.

The direct clone requirement is computed as:

$$DCR(f.m,\ g.n) = OD(MDG(f.m, g.n) + 1$$

For example, the direct clone requirement of method *Processor.initialization* can be computed as:

$DCR(Processor.initialization, VM\_Simu) =$

$$OD(MDG(Proc.init, VM\_Simu)) + 1 = 3.$$

The DCRs of other used methods are given as in Table 4.3. The DCRs for all used methods can be depicted as weights in the MCG (see Figure 4.5). Finding the largest out degree of nodes in a MDG can be done by scanning each node in $O(S)$ (S is number of nodes in the MDG). Therefore, this approach is very fast for calculating the DCR of a used method.

4.4 Transitive Clone Requirements

As discussed in Section 4.2, the use relation of methods is transitive. For example, method *InstrMem.initialize* is directly used by method *Processor.initialization*, and method *Processor.initialization* is directly used by procedure *VM_Simu* as shown in Figure 4.3. Thus, method *InstrMem.initialize* is transitively used by

Table 4.1 The DCRs of methods in the virtual machine simulation.

Synthesized Methods	*Directly Used Methods*	*DCRs*
Processor.dumpDataMemory	DataMem.fetch	1
Processor.doInstruction	DataMem.store	1
	DataMem.fetch	1
	InstrMem.fetch	1
Processor.initialization	DataMem.initialize	1
	InstrMem.store	1
	InstrMem.initialize	1
VM_Simu	Processor.initialization	3
	Processor.doInstruction	3
	Processor.getState	3
	Processor.dumpDataMemory	1

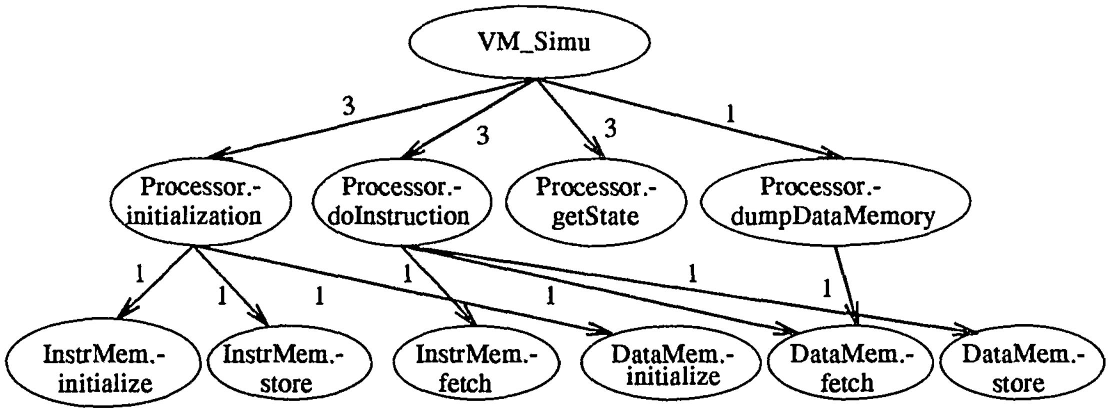


Figure 4.5 The method call graph with DCRs of the virtual machine simulation.

procedure *VM_Simu*. Generally, if a method m exported by an ADT instance f is directly used by a method n exported by an ADT instance g, and a method n is directly used by a method p exported by an ADT instance h ($h \neq g \neq f$), then p is transitively used by m. The number of clones of method m required to resolve all the contention in method p is called the *transitive clone requirement (TCR)* (denoted as $TCR(f.m,\ h.p)$). Note that a method can be both directly and transitively used by another method. The number of clones needed to resolve both direct and transitive contention is called *clone requirement (CR)*. The following discussion shows how to compute the number of clones required for (1) private methods and (2) shared methods.

4.4.1 Clone Requirements of Private Methods

Assume a statement s in a method m calls a method p_1. Assume p_1 is not a primitive method and it further calls method p_2. p_2 calls p_3, and so forth, until a primitive method p_k is called. A calling sequence $p_1 \bullet p_2 \bullet \cdots \bullet p_k$ is formed by this call sequence. The calling sequence stops growing and begins a returning sequence at method p_k. If all the methods on a calling sequence $p_1 \bullet p_2 \bullet \cdots \bullet p_k$ are private methods, the calculation of TCRs of the transitively used methods p_i is the product of DCRs on the calling sequence. The number of clones of p_i needed by m, due to transitive requirements is:

$$TCR(p_i, m) = DCR(p_1, m) \times \prod_{j \in [1, i-1]} DCR(p_{j+1}, p_j)$$

Since all used methods are private, there are no shared methods, i.e., a method is transitively used by only one method, therefore,

$$CR(p_i, m) = TCR(p_i, m)$$

For example, since method *InstrMem.initialize* is a private child of method *Processor.initialization* which is a private child of procedure *VM_Simu*, therefore,

$$\begin{aligned} CR(InstrMem.initialize, VM\_Simu) &= TCR(InstrMem.initialize,\ VM\_Simu) \\ &= DCR(Processor.initialization, VM\_Simu) \\ &\times DCR(InstrMem.initialize, Processor.initialization) \\ &= 3 \times 1 = 3. \end{aligned}$$

All other methods except *DataMem.fetch* are private methods. Similarly, the clone requirements of those methods can be easily calculated as follows.

$$CR(InstrMem.initialize,\ Processor.initialization) =$$
$$DCR(InstrMem.initialize, Processor.initialization) = 1.$$

$$CR(InstrMem.store, Processor.initialization) =$$
$$DCR(InstrMem.store, Processor.initialization) = 1.$$

$$CR(DataMem.initialize, Processor.initialization) =$$
$$DCR(DataMem.initialize, Processor.initialization) = 1.$$

$$CR(InstrMem.initialize,\ VM_Simu) =$$
$$\begin{aligned} &DCR(Processor.initialization, VM_Simu) \\ &\times DCR(InstrMem.initialize, Processor.initialization) \\ &= 3 \times 1 = 3. \end{aligned}$$

CRs for other used methods can be calculated similarly. The CRs are shown as edge weights in the MCG corresponding in Figure 4.6.

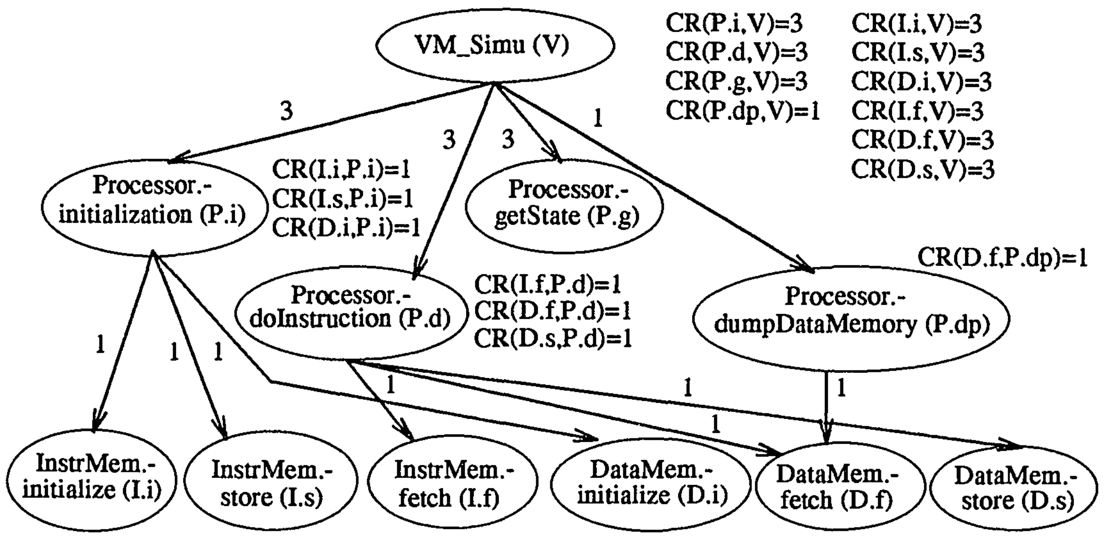


Figure 4.6 The method call graph with CRs of the virtual machine simulation.

4.4.2 Clone Requirements of Shared Methods

When one or more shared methods appear in the calling sequence started at a statement of a method, the calculation of TCRs is not simply the product of DCRs. For example, method *DataMem.fetch*$(D.f)$ is shared by method *Processor.doInstruction*$(P.d)$ and method *Processor.dumpDataMemory*$(P.dp)$ as shown in Figure 4.6. The clone requirements of a method m by a calling sequence $< c_1, c_2, \cdots, c_n >$ is defined as:

$$CR(m, < c_1, c_2, \cdots, c_n, m >) = DCR(m, c_n) \cdot DCR(c_n, c_{n-1}) \cdots DCR(m, c_1)$$

Note that $VM\_Simu(V)$ needs three clones of $P.d$ which needs one clone of $D.f$. Thus, the calling sequence $< V \bullet P.d \bullet D.f >$ has the clone requirement of:

$$CR(D.f, < V \bullet P.d \bullet D.f >) = 3 \times 1 = 3$$

Also V needs one clone of $P.dp$ which needs one clone of $D.f$. Thus, the calling sequence $< V \bullet P.dp \bullet D.f >$ has the clone requirement of:

$$CR(D.f, < V \bullet P.dp \bullet D.f >) = 1 \times 1 = 1$$

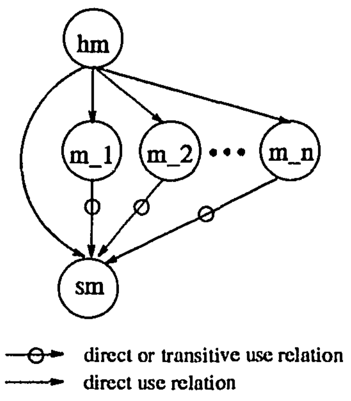


Figure 4.7 Method call graph shows sharing relations.

It may appear that:

$$CR(D.f, VM\_Simu) = CR(D.f, < V \bullet P.d \bullet D.f >)$$
$$+CR(D.f, < V \bullet P.dp \bullet D.f >) = 3 + 1 = 4.$$

However, this is wrong since the statements calling *Processor.dumpDataMemory* and the statements calling *Processor.doInstruction* belong to two different branches of an if statement, they cannot both be executed. Therefore, only three clones of *Processor.doInstruction* could be used concurrently.

If a method n is shared by method p and method q, and both p and q are used by method m, then:

$$n \in TUMS(p) \cap TUMS(q) \vee n \in TUMS(p) \cap DUMS(q) \vee n \in DUMS(p) \cap TUMS(q)$$

The possible precedence relations among the statements in m may make some of the dependence relations ineffective, therefore, the relation between calls to methods p and q must be analyzed when $CR(n, m)$ is calculated. Generally, as shown in Figure 4.7, if a method sm is used (directly or indirectly) among set of methods SML, and each method in SML is directly used by a method hm (sm may be directly used by hm), i.e.,

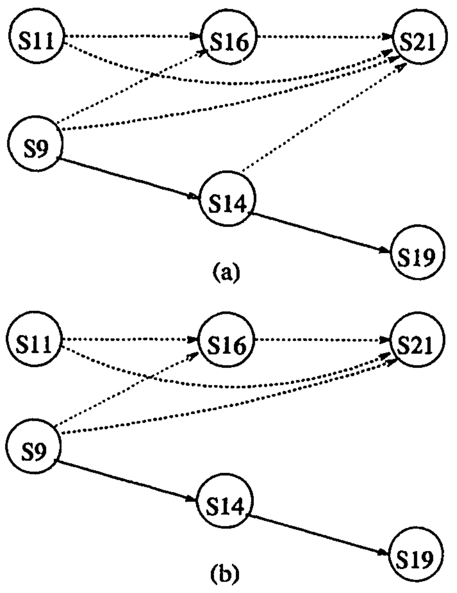


Figure 4.8 (a) $IMDG(\{P.d, P.dp\}, V)$. (b) *IMDG({ P.d, P.dp }, V)* (ineffective dependences removed).

$(\exists sm \in M, hm \in M, SML \subset M)(\forall m \in SML)$

$(DUse(sm, m) \vee TUse(sm, m)) \wedge DUse(m, hm)$

where M is set of methods in an application, $DUse(m1, m2)$ denotes the direct use relation between $m1$ and $m2$, $TUse(m1, m2)$ denotes the transitive use relation between $m1$ and $m2$.

Then, the dependence relations among statements in all methods in SML must be analyzed since the possible precedence relations among the statements may make some of the dependence relations ineffective.

Generally, assume there is a method n which is shared by k ($k > 1$) methods $m_1, m_2, \cdots, m_k$, and methods m_i ($1 \leq i \leq k$) are directly used by another method m (note that n can also be directly used by method m, and n may be directly or indirectly used by $m_i(i = 1, \cdots, k)$). To determine the CR of n by m, the dependence relations among all statements in m which call method m_i ($1 \leq i \leq k$) must be analyzed since method n is used either

```
BuildIMDG(hm, SML, sm)
    /* $SM$ is a method shared among methods in the list $SML$;
    each method in $SML$ is directly used by a method $HM$. */
  begin
    $m$ = get a method from $SML$;
    $IMDG$ = MDG(sm,m);
    while $SML$ is not empty do
      $m$ = get a method from $SML$;
      $oldIMDG = IMDG$;
      $IMDG = IMDG \cup MDG(sm,m)$;
      for each node $n \in MDG(sm,m)$ do
        for each node $p \in oldIMDG$ do
          if $n \rightarrow p$ then
            add $n \rightarrow p$ into $IMDG$;
          else if not $MutualExclusiveRegion(n,p,GDG(m))$ then
            if $label(n) < label(p)$ then
              add $n \rightarrow_{sm} p$ into $IMDG$;
            else
              add $p \rightarrow_{sm} n$ into $IMDG$;
            end if;
          end if;
        end for;
      end for;
    end while;
    $RemoveIneffectiveMD(IMDG)$;
  end
```

Figure 4.9 Algorithm for building an IMDG.

directly or transitively by method m_i ($1 \leq i \leq k$). Code dependence graphs $MDG(m_1, m), MDG(m_2, m), \cdots, MDG(m_k, m)$ are combined together into single code graph called the *inter-method dependence graph (IMDG).* The algorithm for combining MDGs into an IMDG is presented in Figure 4.9. The algorithm combines the MDGs together into IMDG first, precedence relations among nodes in different MDGs are added into the IMDG. Code dependence relations among nodes in different MDGs are checked whether they belong to mutually exclusive region nodes before being put into the IMDG. The function $MutualExclusiveRegion(n, p, GDG(m))$ checks whether nodes n and p belong to two mutually exclusive regions nodes. It is in the most inner loop and takes $O(S)$ in the worst case (S is the number of statements in a method). The inner for loop takes $O(S^2)$. The outer loop takes $O(S^3)$. The while loop takes $O(M \cdot S^3)$ (where M is number of methods in an application). The procedure call $RemoveIneffectiveMD(IMDG)$ takes $O(S^3)$

(discussed in Section 3.4). Therefore, the complexity of the algorithm is $O(M \cdot S^3)$. For the example of the virtual machine simulation, method *DataMem.fetch* is shared by method *Processor.doInstruction* and method *Processor.dumpDataMemory* (as shown in Figure 4.3). To analyze the dependence relation between statements calling method *Processor.doInstruction* and statements calling method *Processor.dumpDataMemory*, MDG(*processor.doInstruction*, *VM_Simu*) (shown in Figure 3.17 (c)) and MDG(*processor.dumpDataMemory*, *VM_Simu*) (shown in Figure 3.17 (d)) are combined into one graph (as shown in Figure 4.8 (a)).

All the nodes in the *IMDG* are related with an indirected code dependence on method n. Thus, undirected dashed edges are added between the nodes in MDGs. Similar to the approach for calculating DCRs, algorithm *RemoveIneffectiveFD* (presented in Figure 3.24) is used to remove all the ineffective code dependence relations from IMDGs. The simplified graph is denoted as:

$IMDG(n\{m_1, m_2, \cdots, m_k\}, m)$.

For the example of the virtual machine simulation, *IMDG*(*DataMem.fetch*{*Processor.doInstruction*, *Processor.dumpDataMemory*}, *VM_Simu*) (after removing all ineffective code dependence relations) is shown in Figure 4.8 (b).

Determining the *OD*(*IMDG*) (the maximum out degree) is similar to finding the *OD*(*MDG*), except that nodes in the *IMDG* may have different clone requirements of the shared method, while nodes in an *MDG* have the same clone requirements. Since nodes in an *IMDG* may call different methods, they may have different CRs for the shared method. Assume nodes $s_1, s_2, \cdots, s_k$ are code dependent on node s. Let $WOD(s, m)$ denote the weighted out degree of node s, where the weight of each $s_i (i = 1, 2, \cdots, k)$ is its CR for n:

$$WOD(s) = \sum_{i=1}^{k} CR(n,\ s_i)$$

In $IMDG(n\{m_1, m_2, \cdots, m_k\},\ m)$, the largest weighted outgoing degree, which is called the weighted out degree of the graph, is denoted as:

```
CalculateCRs (MCG with DCRs ): MCG with CRs
  begin
    $RTO$ = set of nodes in the MCG in reverse topological order;
    $SH = \phi$;
    while $RTO \neq \phi$ do
      $m$ = next node from $RTO$;
      for each $p \in UMS(m)$ do
        for each $q \in DUMS(m)$ do
          if ($p \in UMS(q)$) then
            $SH = SH + q$;
          end if;
        end for;
        if $SH \neq \phi$ then
          $CR(p, m) = OD(IMDG(SH, m)) + 1$
        else
          $CR(p, m) = DCR(p, m)$
      end for
    end while
  end
```

Figure 4.10 Algorithm for calculating clone requirements.

$WOD(IMDG(n\{m_1, m_2, \cdots, m_k\},\ m))$, and is used to calculate the clone requirement of method n by method m:

$$WOD(IMDG(n\{m_1, m_2, \cdots, m_k\},\ m)) = MAX_{s \in S} WOD(s)$$

Then, $CR(n, m)$ can be evaluated as:

$$CR(n, m) = WOD(IMDG(n\{m_1, m_2, \cdots, m_k\},\ m) + 1$$

For the example of the virtual machine simulation, let us calculate the clone requirement of method $DataMem.fetch(D.f)$ by method $VM\_Simu(V)$. Since $CR(D.f, P.d) = 1$ and $CR(D.f, P.dp) = 1$, all the statements in the $IMDG(D.f\{P.d, P.dp\}, V)$ have weight 1. The maximum weighted out degree is at node $S11$. Therefore,

$$\begin{aligned} CR(D.f, V) &= WOD(s_{11}) + 1 \\ &= CR(D.f, P.d) + CR(D.f, P.dp) + 1 \\ &= 1 + 1 + 1 = 3 \end{aligned}$$

Figure 4.10 presents the algorithm for calculating CRs for shared methods. The calculation of CRs is done in reverse topological order of the nodes in the MCG so that by the time the CR of a method is evaluated, all its descendants' CRs are calculated. For each method, the algorithm checks the number of parent methods. If the method is a private method, DCR is used to calculate CR. Otherwise, IMDG is built and the weighted out degree of the IMDG is computed to calculate CR for the shared method. Let M be the number of methods in an application, and S be the maximum number of statements in a method. Topological sorting takes $O(MlogM)$, building IMDG takes $O(M \cdot S^3)$, function call OD for finding the weighted out going degree of an IMDG takes $O(S)$. The inner for loop takes $O(M)$ in worst case. The outer for loop takes $O(M)$ in worst case. The while loop takes $O(M)$ in worst case. Therefore, the worst case complexity of the algorithm is $O(M^3 \cdot S^4)$.

4.5 Cloning Within Loops

The cloning analysis techniques discussed above handle assignment statements, procedure call statements, and conditional statements. For loop statements, the dependence analysis is complicated, since loop statements contain cyclic dependence relations. Backward data dependence relations are caused by the data dependence relations between iterations of loops. To handle loops, we need to consider not only the dependence relations within one iteration, but also the dependence relations across iterations. One simple way to reveal the precedence relations between iterations is to unroll the loop (if the number of iterations is not too large). By unrolling, loop statements are transformed into conditional statements, therefore, the techniques presented in the previous sections can used to perform the dependence and cloning analyses. If the number of iterations is very large, or if loops are unbounded, partial unrolling can be applied, which unrolls only a certain number of iterations (depending on the amount of concurrency needed). Clone analysis id then

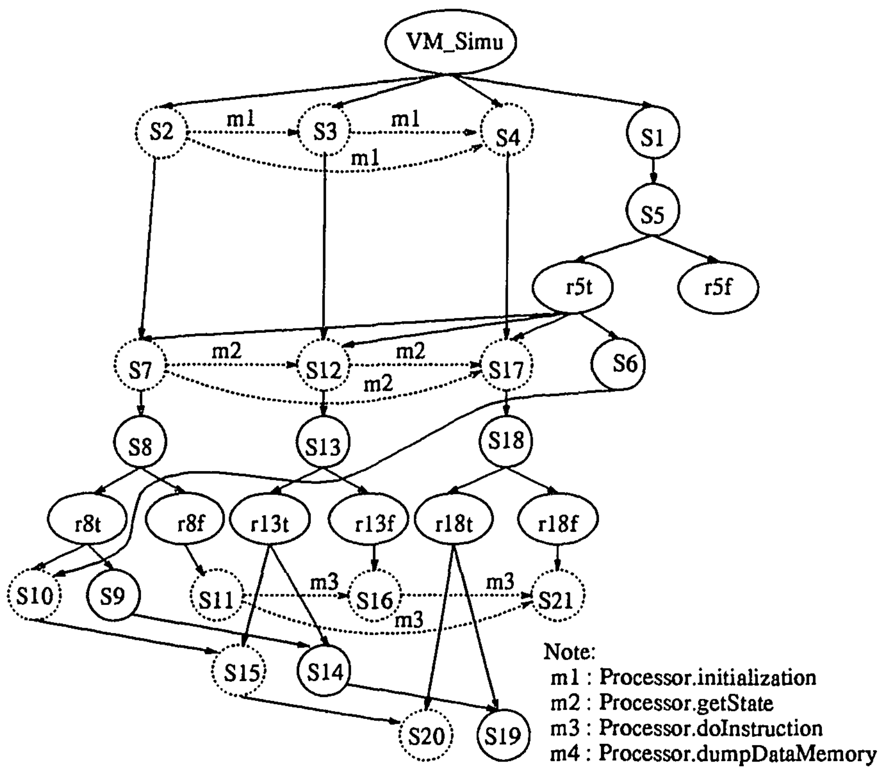


Figure 4.11 The GDG of *VM_Simu* after optimization.

applied ignoring backward dependences. Each time a loop is unrolled, additional cloning opportunities may be revealed by considering only forward dependences.

4.6 Statement Reordering for Exploitation of Concurrency

If a node (a statement node or a region node) in a GDG has more than one child, the children of the node can run in parallel. If one of the children is a local statement, it prevents the following calls from being sent out for concurrent execution. To achieve the maximum concurrency, the children must be reordered so that local statements will not block before all the ARPCs are made.

For instance, the GDG of *VM_Simu* (shown in Figure 3.25) indicates that statements $s1$, $s2$, $s3$, and $s4$ can run in parallel since there is no precedence relation among them (assuming there are three clones of *Processor.initialization* available).

```
Reorder($m$)
  /* $m$ is a method */
  begin
    for each node $N$ that has more than one child in $SPG(m)$ do
      Let $c_1, c_2, \cdots, c_k$ be the childern of $N$;
      $localPtr = 1$; $callPtr = k$;
      while $localPtr < callPtr$ do
        while $c_{localPtr}$ is a call statement do
          $localPtr = localPtr + 1$;
        end while;
        while $c_{callPtr}$ is a local statement do
          $callPtr = callPtr + 1$;
        end while;
        if $localPtr < callPtr$ then
          swap $c_{localPtr}$ and $c_{callPtr}$;
          $localPtr = localPtr + 1$;
          $callPtr = callPtr + 1$;
        end if;
      end while;
    end for;
  end
```

Figure 4.12 Algorithm for statement reordering.

Actually, concurrent execution cannot be achieved since statement $s1$ is the first statement among the four statements and it is a local statement; the ARPCs of $s2$, $s3$, and $s4$ will not be distributed until the finish execution of statement $s1$. To achieve the maximum concurrency, the execution of statement $s1$ can be delayed until after distributing all the ARPCs. By placing the local statements after ARPCs, local statements can be executed while the ARPCs are being processed by other processors. Therefore, maximum concurrency is achieved. In procedure *VM_Simu*, for example, $s1$ blocks ARPCs $s2$, $s3$, and $s4$. $s6$ blocks ARPC $s7$. $s10$, $s15$, and $s20$ blocks ARPCs $s9$, $s14$, and $s19$, respectively. The algorithm for reordering an GDG to allow maximum concurrency is presented in Figure 4.12. It finds all the nodes that are possible ARPCs by checking the number of children of each node in the SPG. Then, sort the children in the order that call the call statements come before all the local statements. The complexity of the algorithm is $O(S^3)$. The GDG after optimization is shown in Figure 4.11.

4.7 Cloning on the ADT Instance Level

The techniques we discussed in the previous sections apply cloning on the method level to achieve concurrency via ARPCs. The product of cloning analysis is a number – the minimum number of clones of each method needed to resolve all possible contention. Cloning on method level leads to more precise upper bounds. However, it sacrifices the high complexity due to the fact that there are very large number of methods in an application. Since methods are defined in ADT modules, ADT instances are usually used as distribution and cloning units to reduce the complexity. The result of cloning analysis on method level can be easily aggregated to ADT instance level to determine the clone requirements of ADT instances. As shown in Figure 4.13, the results of statement level dependence analysis leads to the dependence analysis on method level, and the products of method level analysis are aggregated onto the ADT instance level.

An *instance call graph* (ICG) is built for showing the use relations among the ADT instances in the application. The nodes in the ICG represent ADT instance or global procedure. The directed edges in the ICG represent use relations. If the statements in the methods exported by an ADT instance I call the methods exported by another ADT instance J, instance J is said to be a used instance of I and the use relation is represented with a directed edge from node I to node J in the ICG. For example, the virtual machine simulation has three ADT instances ($Processor$, *DataMem*, and $InstrMem$) and one global procedure (VM_Simu); its use relations are shown in the ICG of Figure 4.14.

Figure 4.17 shows the algorithm for determining CRs at instance level. The aggregation of CRs from method level to instance level is done by first calculating CR of each ADT instance separately. For each ADT instance, the maximum CR among the CRs of the used methods of the instance is used as the initial CR value of the instance, and all the nodes in the MDG of the method that has the maximum

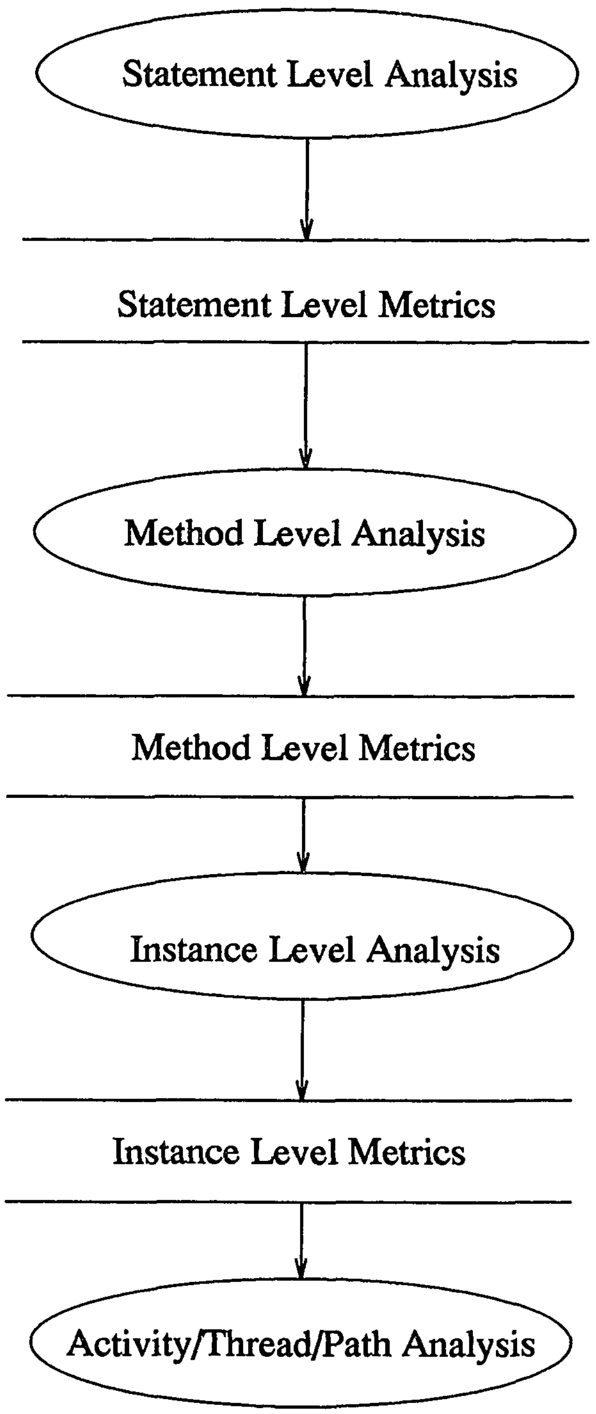


Figure 4.13 The aggregation to larger grains of programs.

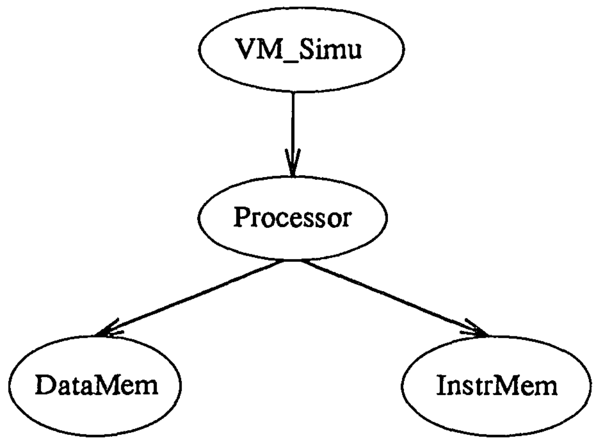


Figure 4.14 The instance call graph of virtual machine simulation.

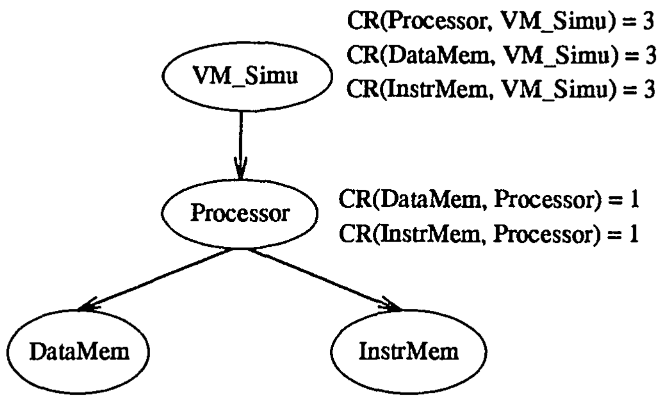

CR(Processor, VM_Simu) = 3
CR(DataMem, VM_Simu) = 3
CR(InstrMem, VM_Simu) = 3
VM_Simu
Processor
CR(DataMem, Processor) = 1
CR(InstrMem, Processor) = 1
DataMem
InstrMem

Figure 4.15 The ICG of virtual machine simulation with CRs.

CR are marked as *checked.* For each statement in the MDGs of other methods of the instance, if the statement has neither a precedence relation with the *checked* calls, nor they belong to mutual exclusive regions, the CR of the instance is incremented by one, and the call is marked as *checked.* This is because the node in other MDGs calls a different method, but the same instance, if the statement has no precedence relation with any of the *checked* nodes, then it can execute concurrently. For example, to determine $CR(Processor, VM\_Simu)$, find the largest CR among the CRs of methods exported by instance $Processor$. Recall that $CR(Processor.initialization, VM\_Simu) = 3$, $CR(Processor.doInstruction, VM\_Simu) = 3$, $CR(Processor.getState, VM\_Simu) = 1$, and $CR(Processor.dumpDataMemory, VM\_Simu) = 3$, Therefore, initially, $CR(Processor, VM\_Simu) = 3$, and the $checkedSet = \{s2, s3, s4\}$. Since the other call statements $s7, s9, s11, s12, s14, s16, s17, s19, s21$ in VM_Simu all have precedence relations with the statements in the *checkedSet*, therefore, $CR(Processor, VM\_Simu) = 3$. Similarly, other clone requirements can be determined as shown the Figure 4.15. After cloning, the ICG which describes the call relations among ADT instances and clones of ADT instances is shown in Figure 4.16. Three clones of $Processor$ manage three PEs, respectively. Each clone of $Processor$ has its own clones of $MemoryManger$ to manage its data and instruction memories. The complexity of the algorithm is $O(I \cdot S^2)$ (I is the largest number of methods in an ADT module).

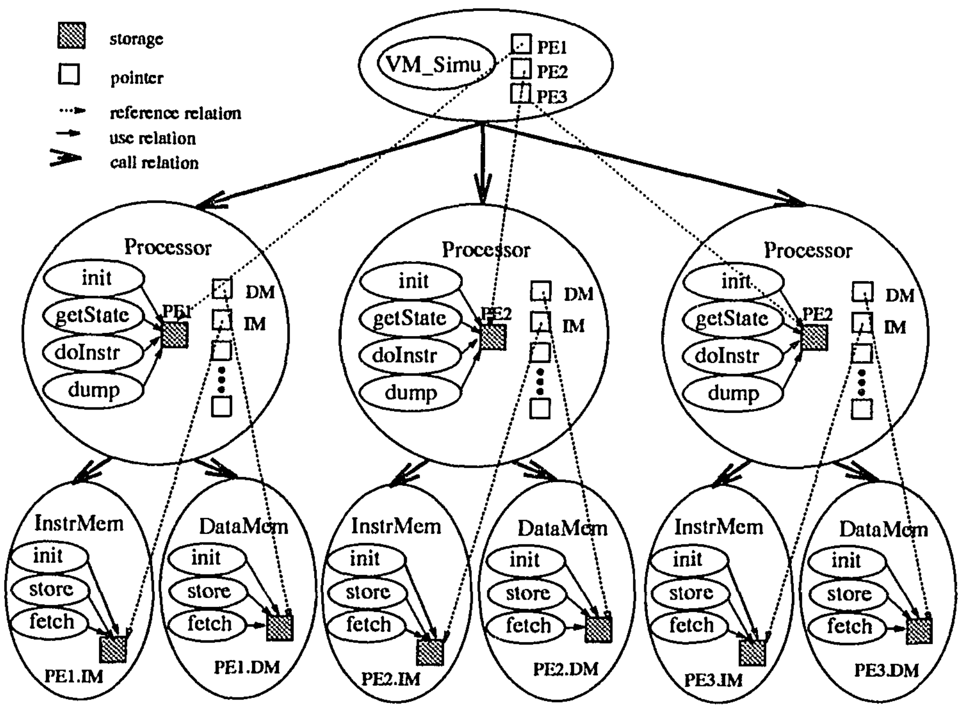


Figure 4.16 The ICG of virtual machine simulation with clones of instance.

```
determineInstanceCRs (instance I1, I2)
  begin
    Let the ADT module of the instance I2 be A;
    Let the ADT module of the instance I1 be B;
    Let $M_1, M_2, \cdots, M_k$ be the methods of A;
    Let $m1, m2, \cdots, ml_i$ be the used methods exported by B;
    $CR(I1, I2) = 0$;
    for each method $M \in M_1, M_2, \cdots, M_k$ do
      $checkedSet$ = null;
      for each call statement $s$ in $M$ do
        $newCR(I1, I2) = MAX(CR(m1, M), CR(m2, M), \cdots, CR(ml_i, M))$;
        Let $mm$ denote the method that has the maximum CR;
        $checkedSet$ = all the statements that call method $mm$;
        if $s \notin checkedSet \wedge \forall s' \in checkedSet(s' \not\to s \wedge s \not\to s')$ then
          $newCR(I1, I2) = newCR(I1, I2) + 1$;
        end if;
        enter $s$ into $checkedSet$;
      end for;
      if $newCR(I1, I2) > CR(I1, I2)$ then
        $CR(I1, I2) = newCR(I1, I2)$
      end if;
    end for;
  end
```

Figure 4.17 The algorithm for determining CRs at the instance level.

CHAPTER 5

INCREMENTAL PARALLELIZATION

In hard real-time systems, the most important goal is to guarantee, either by schedule construction or by analysis, that all timing constraints are satisfied. Scheduling problems are NP-hard in multiprocessor systems [57]. On-line scheduling approaches are not typically sufficient to guarantee timeliness, due to the limited amount of time for scheduling and the overhead of optimal scheduling. However, on-line scheduling techniques [36, 39, 45, 80, 46] are necessary in applications that have unpredictable environments. For a fully predictable or almost fully predictable environment, off-line scheduling techniques are used to guarantee timeliness. Contention for shared resources (processors, devices, and communications) is avoided by constructing schedules before runtime for each shared resource. If there exist a few unpredictable factors, several schedules are constructed. At run-time, one of the schedules is chosen. Off-line scheduling is being used successfully in many application areas, including factory automation, telecommunication, aerospace, and robotics [58, 18, 63].

Traditional off-line scheduling approaches try all possible permutations of the *scheduling objects* (processes, tasks, segments of processes, etc.) to seek a feasible solution. The timing behavior of scheduling objects is unchanged during scheduling, thus all effort is devoted to optimizing the search path for finding feasible schedules [44, 74, 61, 59, 60, 30, 37, 49]. This chapter presents a new approach for constructing off-line schedules for applications composed of abstract data type (ADT) modules. This approach constructs an initial schedule based on sequential execution. The initial schedule is evaluated to see if all the timing constraints are satisfied and if it can be improved. If the schedule needs to be improved and can be improved, the critical path and a list of critical methods are identified. Candidates are evaluated by analyzing effects on the entire schedule, the utilization and availability of demanded resources, and the amount of concurrency that can be produced if parallelization is applied. The best candidate is chosen to improve the assignment and schedule. The execution time of scheduling objects is reduced by enhancing concurrency. The chance of finding a feasible schedule is significantly increased by concurrency enhancement, since processing resources can be employed to decrease execution times of processes missing deadlines.

The assumptions of the scheduling techniques are:

- only CPU of each PE is scheduled;
- infinite number of PEs are available;
- cloning is applied only to stateless ADTs;
- the parallelization techniques are applied at single task with deadline only;
- a fast communication network is available.

As shown in Figure 1.1, there are three major steps of the scheduling approach: initial schedule construction; identifying critical methods; and parallelizing a critical method. The last two steps are repeated until the schedule is made feasible or there is no more chance for exploiting concurrency. The top level schedule algorithm is presented in Figure 5.1. Constructing statement table is discussed in Section 3.2.1. Statement table is tree describing the control flow, thus, a CFG can be constructed directly with the statement table (as discussed in Section 3.2.1). Construction of MCG and ICG is done by the parser using the statement table (as discussed in Section 4.2). Initial schedule construction is discussed in Section 5.1. Dependence analysis is presented in Chapter 3. Cloning analysis is presented in Chapter 4. Agreggating clone requirements from method level to instance level is discussed in Section 4.7. Identifying critical paths and critical methods is discussed in Section 5.2.

Estimating concurrency and communication cost is presented in Section 5.3. Parallelizing a critical method is discussed in Section 5.5.

```
IncrementalScheduling($Task$);
  begin
    $StaTab$ = ConstructStatementTable($Task$);
    ConstructCFG($StaTab$);
      /* construct a CFG for each method */
    ConstructCallGraph($StaTab$);
      /* construct MCG, ICG for $Task$ */
    $Sch$ = ConstructInitialSchedule($Task$, $MCG$);
    numPEUsed = 1;
    if $Task$ misses deadline then
      DependenceAnalysis($Task$);
        /* construct a CDG, DDG, PDG, SPG, GDG, MDG, and IMDG for each method */
      CloningAnalysis($Task, MCG$);
        /* calculate DCR and CR for each method */
      AgreggateToInstanceLevel($Task$, $ICG, DCRs, CRs$);
        /* calculate DCR and CR for each ADT instance*/
      $V_{Topo}$ = sortInTopoOdr(MCG);
      while $Task$ misses deadline in $Sch$ and exists unparallelized CM do
        A = getNextNode($V_{Topo}$);
        $CMList$ = IdentifyCriticalMethod($A$);
          /* estimating concurrency and communication overhead
        $Max = 0$;
        $CM_{max}$ = null;
        for $CM \in CMList$ do
          /* mode can be SEQ, SRPC, ARPC, or ARPC&CLONING */
          $CC(CM)$ = estimatingCommunicationCost($CM, numBuses, mode$);
          for each call $s$ to $CM$ in $A$ do
            $OET(s)$ = estimatingOverlappingTime($s, Task, mode$);
            if $Max < OET(s)/CC(s)$ then
              $CM_{max} = CM$;
              $Max = OET(s)/CC(s)$;
            end if;
          end for;
        end for;
        $Sch$ = ParallelizingCriticalMethod($Task, CM_{max}, mode$);
      end while;
    end if;
  end;
```

Figure 5.1 An incremental scheduling algorithm.

This chapter is organized as follows: Section 5.1 presents the approach for initial schedule construction. Section 5.2 introduces the way for identifying critical path and methods. Section 5.5 discusses the parallelization of critical methods. Finally, the virtual machine simulation is used to illustrate the schedule approach.

5.1 Initial Schedule Construction

The initial schedule is constructed based on sequential execution. If the initial schedule can not meet the time requirements, concurrency will be enhanced via ARPCs and ADT instance cloning. The algorithm for constructing an initial schedule is presented in Figure 5.2. It computes execution times of primitive methods first. Then a method call graph is constructed. Following the reverse topological order of the method call graph, it constructs the initial schedule layer by layer.

```
ConstructInitialSch ($Task, MCG$)
  begin
    $SchLength = 0$;
    $V_{RevTopo}$ = sortInRevTopoOdr($MCG$);
    for $I \in V_{RevTopo}$ do
      if $I$ is primitive method then
        begin
          Compute execution time of a primitive method;
        end;
      else if $I$ is synthesized method then
        begin
          $CFG(I)$ = evaluateExecutionTime($CFG(I)$);
            /* EST and LST of each statement in $I$ is determined,
            and stored in CFG of $I$ */
          $Sch(I)$ = constructLocalSch($CFG(I), SchLength$);
        end;
      end if;
    end for;
  end;
```

Figure 5.2 Algorithm for constructing an initial schedule.

To determine if execution times are met, we compute tasks execution times. To evaluate the execution time of a task, the execution times of the methods it calls must be known. Recall that the call relations among methods are described in an MCG. In an MCG, leaves represent primitive methods. Internal nodes represent synthesized methods (they call other methods). The execution times of primitive methods can be evaluated in instruction cycles since they contain only local instructions. By following the reverse topological order of the MCG, the execution times of synthesized methods can be calculated based on the execution times of methods at lower levels. The algorithm for evaluating the execution time of a method is presented in Figure 5.3. With the CFG of a method as a parameter to the algorithm, it calculates the finish time of each statement in a topological order of the CFG. The parent-child relation in a CFG is a dependence relation due to control flow. By following the topological order, the finish times of parent nodes are calculated before calculating the finish times of child nodes, i.e., a statement cannot start execution until all its parent statements are finished. If a node in a CFG has only one parent, the finishing time of the node is the finishing time of its parent plus its own execution time. If a node has more than one parent, its finish time is the finish time of the parent that takes longest to finish plus its own execution time. Finally, the execution time of the method is the finish time of the leaf node (statement) that takes longest to finish.

To create a schedule of a method based sequential execution, a CFG of the method is used since a CFG of a method describes the control flow of sequential execution of the method. Figure 5.4 presents the algorithm for constructing the initial local schedule based on sequential execution of a method.

Since the statement table of a method is a tree of sub-statement tables, the construction of a CFG of a method ($constructCFG$) can be done in $O(S)$ (S is the number of statements in the method). Constructing the local schedule of a method ($constructLocalSch$) based on the CFG of the method can be done in $O(S)$. Evaluating the execution time ($evaluateExecutionTime$) takes $O(S)$. Therefore, The time complexity of the algorithm $ConstructInitialSch$ is $O(M \cdot S)$, where M is the number of method in an application, and S is the maximum number of statements in a method. Therefore, the complexity of the local schedule construction is $O(M^2 \cdot S)$.

For example, Figure 5.5 shows the CFG of procedure VM_Simu. The nodes with dotted circles are call statements. Each node is labeled with the execution time of the statement. This CFG is the base for constructing the initial schedule (the

```
evaluateExecutionTime ($CFG$);
  begin
    Let $PO$ = the topological order of the nodes in CFG;
    $R^0_{bft} = 0$ and $R^0_{wft} = 0$.
    \* $R^0$ (entry node of method $M$) starts at time (0,0)
    This entry node is served as a base point, all other nodes'
    finishing times are relative to this entry node. *\
    For each node $N^i \in PO$ do
      if $N^i$ has only one parent node $N^p$, then
        $N^i_{bft} = N^p_{bft} + N^i_{bet}$
        $N^i_{wft} = N^p_{wft} + N^i_{wet}$
        \* A parent-child relation describes a forced execution order.
        It can be data precedence relation (as the edge in a SPG)
        or control flow relation (as the edge in a CFG),
        Therefore, the finishing time of child node is equal to its
        parent node's finishing time plus its execution time.*\
      end if
      if $N^i$ has more than one parent node $N^p$ $(p = p1, p2, ..., pn)$, then
        $N^i_{bft} = MAX(N^{p1}_{bft}, N^{p2}_{bft}, ..., N^{pn}_{bft}) + N^i_{bet}$
        $N^i_{wft} = MAX(N^{p1}_{wft}, N^{p2}_{wft}, ..., N^{pn}_{wft}) + N^i_{wet}$
        \* If a node has more than parent node,
        the earliest starting time of the child node is after
        all its parent nodes finished, i.e., it must wait
        for the parent node which takes longest to finish.
        Therefore, The $MAX$ operation is used. *\
      end if
    end for
    \* The execution time of the method is the earliest
    time at which all statements in the method finished.
    It is equal to the finishing time of the leaf node that takes
    longest to finish among all leaf nodes. *\
    $M_{bet} = MAX(LN^1_{bft}, LN^2_{bft}, ..., LN^l_{bft})$
    \* The best case execution time of method $M$ *\
    $M_{wet} = MAX(LN^1_{wft}, LN^2_{wft}, ..., LN^l_{wft})$
    \* The worst case execution time of method $M$ *\
  end
```

Figure 5.3 Algorithm for evaluating execution time of a method.

schedule is too large to be encluded in this thesis) for *VM_Simu*. The execution time of the schedule is 4483 time units. Assume the execution time of *VM_Simu* is 4000 time units. Thus, the initial schedule does not meet the deadline of *VM_Simu*.

5.2 Identifying Critical Methods

Parallelizing methods randomly may not improve the performance quickly. Parallelization should be applied to only the methods that are critical to the performance

```
constructLocalSch($CFG(I), SchLength$)
  begin
    $V_{topoOdr}$ = genTopo($CFG$);
    for $n \in V_{topoOdr}$ do
      if EST(n) = LST(n) then
        Let $n$ start at $EST(n) + SchLength$;
        $SchLength = SchLength + ET(n)$;
      else
        Let $n$ start at $EST(n) + SchLength$;
      end if;
    end for;
  end;
```

Figure 5.4 Algorithm for constructing a local schedule for a method.

of programs. This section introduces the approach for identifying such critical paths, and critical methods.

A critical path in a method is the execution path that takes the longest to finish (a method may have more than one critical path). All the methods called by the statements on the critical paths are called *critical methods.* Obviously, parallelizing the critical methods can reduce the execution time of the method via ARPCs and cloning. When one critical method is parallelized, the execution times of all the calls to that method are ideally reduced. Note that the critical path is changed every time a critical method is parallelized, i.e., previous critical methods may no longer be critical. Therefore, the critical paths and set of critical methods must be recomputed every time a critical method is parallelized.

The algorithm for identifying critical paths and critical methods is presented in Figure 5.6. The algorithm finds the earliest possible start time for each statement in a method by following the topological order of the statements of the method. The execution time of the method is calculated by finding the node that starts latest. The latest possible start time of each statement in the method is calculated by following the reverse topological order of statements of the method. By comparing the earliest

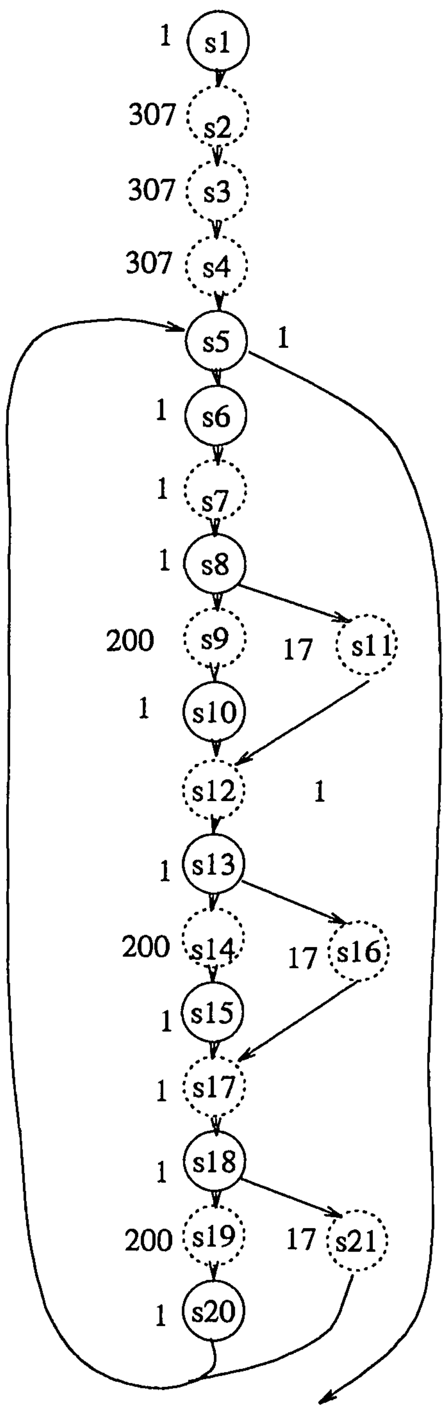


Figure 5.5 The initial CFG of procedure *VM_Simu*.

possible start time with the latest possible start time of each statement, the critical methods are determined.

For example, Figure 5.7 (a) shows a dependence graph with an execution time labeled on each node. ET denotes execution time. EST denotes the earliest start time. LST denotes latest start time. To calculate the earliest start time of each node, a topological order is followed, i.e., $s1, s2, s3, s4, s5$. The earliest start time of $s1$ is zero since it is the first node. Since $s1$ takes 2 time units to finish, $s2$ and $s3$ can start as early as 2. To calculate the earliest possible start time of $s4$, the later finish time of $s2$ and $s3$ is used. Since $s2$ starts at 2 and finishes at 4, and $s3$ starts at 2 and finishes at 5, therefore, the earliest start time of $s4$ is 5. $s3$ starts at 2 and takes 3 time units, so $s5$ can start as early as 5. The shortest possible execution time of the graph is 10 time units. The calculation of latest start times is done in a reverse topological order. One of the reverse topological order is: $s5, s4, s3, s2, s1$. The latest start time of $s5$ is 6 ($ET - ET(s5)$). The latest start time of $s4$ is 5. To calculate the latest start time of $s3$, both $s4$ and $s5$ must be considered. The smaller latest start time of $s4$ and $s5$ is used to calculate the latest start time of $s3$ which is 2 ($LST(s4) - ET(s3)$). The latest start time of $s2$ is calculated by $LST(s4) - ET(s2)$ $= 3$. The smaller latest start time of $s2$ and $s3$ is used to calculate the latest start time of $s1$ which is 0 ($LST(s3) - ET(s1)$). The critical path is thus $s1, s3, s4$. The critical methods are methods called by the statements on the critical path.

For the virtual machine simulation, the critical path of the initial schedule is: $s_1 - s_{10}, s_{12} - s_{15}, s_{17} - s_{20}$. The call statements on the critical path is: $s_2, s_3, s_4, s_7, s_9, s_{12}, s_{14}, s_{17}, s_{19}$. The critical method set is: *initialization*, *GetState*, *dumpDataMemory*, and *doInstruction* exported by *Processor*.

The time complexity for topological sorting is $O(SlogS)$ (where S is the number of statements in a method). The time complexity for calculating the earliest start times is $O(S^2)$. The time complexity for calculating the execution time of a method is

```
IdentifyCriticalMethod(DG)
    /* DG is a dag representation of a method,
    DG can a CFG, PDG, GDG of a method */
  begin
    Let TopoOdr = the topological order of the nodes in DG;
    for each node N of TopoOdr do
      $EST(N) = 0$;
          the earliest start time of node N
      for each node M ($M \neq N$) do
        if $M \rightarrow N$ and $EST(M) + ET(M) > EST(N)$ then
          $EST(N) = EST(M) + ET(M)$ ;
        end if;
      end for;
    end for;
    ET = 0;
    for each node N of TopoOdr do
      if $EST(N) + ET(N) > ET$ then
        $ET = EST(N) + ET(N)$;
      end if;
    end for;
    Let RevTopoOdr = the reverse topological order of the nodes in DG;
    for each node N of RevTopoOdr do
      $LST(N) = ET - ET(N)$;
          the latest start time of node N
      for each node M ($M \neq N$) do
        if $N \rightarrow M$ and $LST(M) - ET(M) < LST(N)$ then
          $LST(N) = LST(M) - ET(M)$ ;
        end if;
      end for;
    end for;
    CriticalMethodSet = null;
    for each node N of TopoOdr do
      if $EST(N) = LST(N)$ then
        put N into CriticalMethodSet;
      end if;
    end for;
  end;
```

Figure 5.6 Algorithm for identifying critical paths and methods.

$O(S)$. The time complexity for calculating the latest start times is $O(S^2)$. The time complexity for finding the critical path and critical method set is $O(S)$. Therefore, the time complexity of the algorithm is $O(S^2)$.

5.3 Estimating Communication Overhead

When parallelization is applied to a method, concurrency is achieved, but communication overhead may be produced too. The communication overhead, varies in different execution paradigms, is explained below.

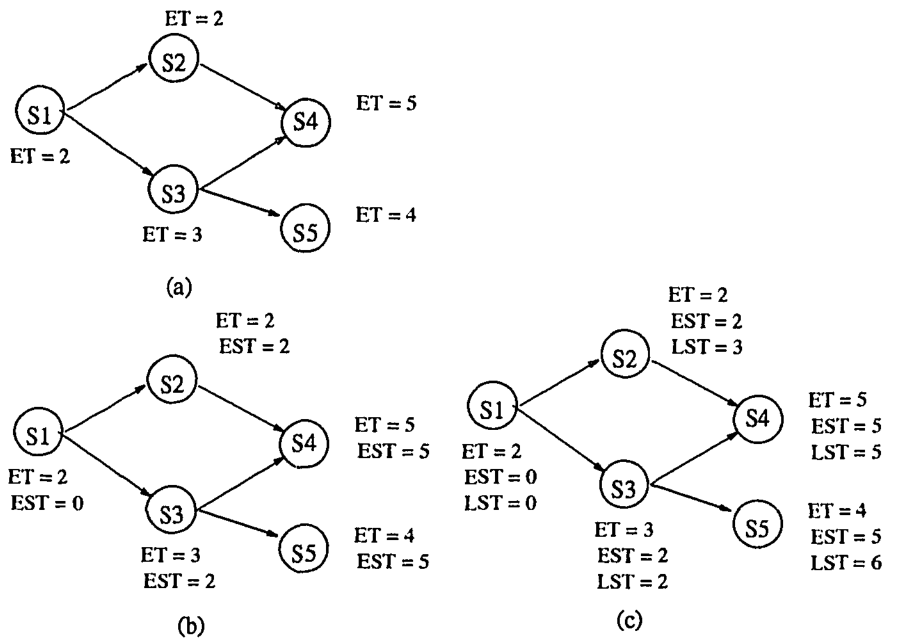


Figure 5.7 (a) dependence graph with ETs. (b) Dependence graph with ESTs. (c) Dependence graph with LSTs.

- In uniprocessor execution, concurrency does not cause any communication cost.
- In the SRPC paradigm, communication overhead produced is the time delay for transmitting remote procedure calls and returns.
- In the ARPC paradigm, the communication overhead produced is not only the the time delay for transmitting remote procedure calls and returns, but also the time delay due to the contention due to multiple remote procedure calls and returns.
- In the ARPC and ADT cloning paradigm, the communication overhead produced is the the time delay for transmitting remote procedure calls and returns, but also the time delay due to the contention caused by the multiple remote procedure calls and returns. Compared with ARPC paradigm, this

paradigm produces more overhead due to the fact that ADT cloning resolves code contention but requires more ARPCs.

The factors that cause the communication overhead are:

- the size of the parameter list of a call to a method m (denoted as $P(m)$);
- the time for packing and unpacking a communication packet for a call to or a return from a method m (denoted as $H(m)$);
- the number of calls to a method m (denoted as $N(m)$), and
- the number of remote procedure calls in method m that can execute concurrently (denoted as $S(m)$);

The communication overhead produced by parallelizing a method m is estimated as:

$$C(m) = \sum_{i=1}^{n} 2((\alpha P(m_i) + H(m_i) + C(m_i)) \cdot N(m_i) + \beta \cdot S(m))$$

where:

1. n is the total number of used methods by method m.
2. $m_i (i = 1 \ldots n)$ is a method called by method m.
3. α is a factor that scales the parameter size and packet handling delay ($\alpha = 0.5$ for this experiment).
4. β is a factor that scales the number of concurrent ARPCs. ($\beta = 0.5$ for this experiment).

In the following subsections, the communication overhead produced by each of the execution paradigms are discussed, and formulas are presented for estimating the communication overhead in each of the paradigms.

5.3.1 Communication in SRPC Paradigm

Assume there is only one connection between any pair of processors, and each method is assigned onto a different processor. Remote procedure calls execute sequentially (i.e., $S(m) = 0$). Since there is no method call in primitive methods, therefore, the communication overhead is zero (i.e., $C(m) = 0$). For synthesized method, the formula for estimating the communication cost caused by the execution of a method m via SRPCs is:

$$C(m) = \sum_{i=1}^{n} 2((\alpha P(m_i) + H(m_i)) + C(m_i)) \cdot N(m_i))$$

The formula calculates the total communication cost of all the calls to other methods. The calculation is followed in the reversed topological order of the method call graph so that the communication cost of each of its used methods is calculated first. For example, in the virtual machine simulation, methods exported by ADT instances *InstrMem* and *DataMem* are primitive methods, therefore, the communication cost of methods exported by instances *InstrMem* and *DataMem* is zero. Method *getState* exported by ADT instance *Processor* is also a primitive method, thus, it communication cost is zero. Method *Initialization* exported by ADT instance *Processor* is a synthesized method, The communication cost of *Processor.Initialization* in SRPC paradigm is:

$$C(\mathit{initialization}) = 2(2+2)/2 + 2((3+2)/2)25 = 129 \textit{ time units.}$$

Method *doInstruction* is a synthesized method exported by instance *Processor*, its communication cost in SRPC paradigm is:

$$C(\mathit{doInstruction}) = 2(3+2)/2 = 5 \textit{ time units.}$$

Method *dumDataMemory* is a synthesized method exported by instance *Processor*, its communication cost in SRPC paradigm is:

$$C(\mathit{doInstruction}) = 2((3+2)/2)50 = 125 \textit{ time units.}$$

Procedure *VM_Simu* is a top level procedure, its communication cost in SRPC paradigm is:

$$C(VM\_Simu) = 4480 \textit{ time units.}$$

5.3.2 Communication in ARPC Paradigm

Assume there is only one connection between any pair of processors, and each method is assigned onto a different processor. Since there is no method call in primitive methods, therefore, the communication overhead is zero (i.e., $C(m) = 0$). For synthesized method, the communication cost caused by the execution of a method m with ARPCs is evaluated using the following formula:

$$C(m) = \sum_{i=1}^{n} 2((\alpha P(m_i) + H(m_i) + C(m_i)) \cdot N(m_i) + \beta B(GDG(m)))$$

where $B(GDG(m))$ is the average number of sibling calls in the GDG of method m.

The sibling calls in the GDG have no dependence relations among them, thus, they can be executed at the same time via ARPCs, they will compete for communication (a queue exists), thus, the number of sibling calls is used to estimate communication contention. The calculation is performed in reverse topological order of the method call graph, so that the communication cost of each of its used methods is calculated first.

For example, in the virtual machine simulation, methods exported by ADT instances *InstrMem* and *DataMem* are primitive methods, therefore, the communication cost of methods exported by instances *InstrMem* and *DataMem* is zero. Method *getState* exported by ADT instance *Processor* is also a primitive method, thus, it communication cost is zero. Method *Initialization* exported by ADT instance *Processor* is a synthesized method, The communication cost of *Processor.Initialization* in ARPC paradigm is:

$$C(initialization) = 2(2+2)/2 + 2((3+2)/2)25 = 129 \textit{ time units.}$$

Method *doInstruction* is a synthesized method exported by instance *Processor*, its communication cost in ARPC paradigm is:

$$C(doInstruction) = 2(3+2)/2 = 5 \textit{ time units.}$$

Method *dumDataMemory* is a synthesized method exported by instance *Processor*, its communication cost in ARPC paradigm is:

$$C(doInstruction) = 2((3+2)/2)50 = 125 \textit{ time units.}$$

Note that for the above three method, the communication cost in ARPC paradigm is the same as in SRPC paradigm. This is because all calls have data dependence relations, the number of sibling calls is zero. Procedure *VM_Simu* is a top level procedure, its communication cost in ARPC paradigm is:

$$C(VM\_Simu) = 2047.5 \textit{ time units.}$$

5.3.3 Communication in ARPC and ADT Cloning Paradigm

Assume there is only one connection between any pair of processors, and each method is assigned onto a different processor. Since there is no method call in primitive methods, therefore, the communication overhead is zero (i.e., $C(m)= 0$). For synthesized method, the communication cost caused by the execution of a method via ARPCs and ADT cloning is evaluated using the following formula:

$$C(m) = \sum_{i=1}^{n} 2((\alpha P(m_i) + H(m_i) + C(m_i)) \cdot N(m_i) + \beta B(SPG(m)))$$

$B(SPG(m))$ is the average number of sibling calls in the SPG of method m.

With full cloning (i.e., each method is cloned up to the upper bound), all the possible contention for method is resolved, then the SPG, which describes only precedence relations, is used to evaluate contention for communication. The calculation is also followed in the reverse topological order of the method call graph so that the communication cost of each of its used methods is calculated first.

For example, in the virtual machine simulation, methods exported by ADT instances *InstrMem* and *DataMem* are primitive methods, therefore, the communication cost of methods exported by instances *InstrMem* and *DataMem* is zero. Method *getState* exported by ADT instance *Processor* is also a primitive method, thus, it communication cost is zero. Method *Initialization* exported by ADT instance *Processor* is a synthesized method, The communication cost of *Processor.Initialization* in ARPC and ADT cloning paradigm is:

$$C(initialization) = 2(2+2)/2 + 2((3+2)/2)25 = 129 \textit{ time units.}$$

Method *doInstruction* is a synthesized method exported by instance *Processor*, its communication cost in ARPC and ADT cloning paradigm is:

$$C(doInstruction) = 2(3+2)/2 = 5 \textit{ time units.}$$

Method *dumDataMemory* is a synthesized method exported by instance *Processor*, its communication cost in ARPC and ADT cloning paradigm is:

$$C(doInstruction) = 2((3+2)/2)50 = 125 \textit{ time units.}$$

Note that for the above three method, the communication cost in ARPC and ADT cloning paradigm is the same as in SRPC paradigm. This is because all calls have data dependence relations, the number of sibling calls is zero. Procedure *VM_Simu* is a top level procedure, its communication cost in ARPC paradigm is:

$$C(VM\_Simu) = 2753.5 \textit{ time units.}$$

Note that communication cost of procedure *VM_Simu* in ARPC and ADT cloning is larger than in ARPC. This is because the ADT cloning produces more contention for communication.

5.3.4 Varying the Interconnection Topology

The above evaluation formulas assume that there is only one connection between any pair of processors. If two connections are available between any pair of processors, The worst case communication cost can be estimated as:

$$C_2(m) = (1 - \frac{1}{4}) * C(m)$$

Note that an additional connection between two processors typically cannot cut the communication costs in half. If there are n connections between any pair of processors, the worst case communication cost is calculated as:

$$C_n(m) = (\frac{3}{4})^{n-1} * C(m)$$

For example, with 2-network, the communication cost $C(VM\_Simu)$ can be reduced to $C_2(VM\_Simu) = (\frac{3}{4}) * C(VM\_Simu) = 2065.1$. With 5-network, the communication cost $C(VM\_Simu)$ can be reduced to $C_5(VM\_Simu) = (\frac{3}{4})^4 * C(VM\_Simu) = 871.2$.

The algorithm for estimating communication overhead is presented in Figure 5.8. It calculates the communication cost for 1-network according to the different execution models, then scales it down with the actual network topology used. Calculating the parameters for the formulas takes $O(S)$. The complexity of the algorithm is $O(S^2)$.

5.4 Estimating Concurrency

Each critical method is evaluated for the amount of potential concurrency. The amount of concurrency is measured by finding the overlapping execution time if it is parallelized. As shown in Figure 5.9, the algorithm finds the predecessor and successor for a call statement according to different execution modes. Then, it calculates the overlapping time by summing up all the execution times

```
EstimatingCommunicationOverhead($CM, numBuses, mode$);
  begin
    /* let $P(m)$ denote the size of parameter list of method $m$ */
    /* let $H(m)$ denote the time for handling a packet of method $m$*/
    /* let $N(m)$ denote the number of calls to method $m$ from method $CM$ */
    /* let $B(GDG(CM))$ denote the average number of sibling calls in the GDG of $CM$ */
    /* let $B(SPG(CM))$ denote the average number of sibling calls in the SPG of $CM$ */
    $CC(CM) = 0$;
    if $CM$ is not primitive method then
      case mode = SEQ :
        $CC(CM) = 0$;
      case mode = SRPC :
        $CC(CM) = 0$;
        for each called method $m$ by $CM$ do
          $CC(CM) = CC(CM) + 2(0.5 \cdot P(m) + H(m)) + CC(M) \cdot N(m))$
        end for;
      case mode = ARPC :
        $CC(CM) = 0$;
        for each called method $m$ by $CM$ do
          $CC(CM) = CC(CM) + 2(0.5 \cdot P(m) + H(m)) + CC(M) \cdot N(m)) + 0.5B(GDG(CM))$
        end for;
      case mode = ARPC&CLONING :
        $CC(CM) = 0$;
        for each called method $m$ by $CM$ do
          $CC(CM) = CC(CM) + 2(0.5 \cdot P(m) + H(m)) + CC(M) \cdot N(m)) + 0.5B(SPG(CM))$
        end for;
      end case;
      $CC(CM) = (\frac{3}{4})^{numBuses-1} * CC(CM)$;
    end if;
    return $CC(CM)$;
  end;
```

Figure 5.8 Algorithm for estimating communication overhead.

of the statements that can continue execution following an ARPC. Functions *findPredecessor* and *findSuccessor* take $O(S)$ (S is the number of statement in a method). The loop for evaluating the overlapping execution time takes $O(S)$ too. Therefore, the complexity of this algorithm is $O(S)$.

5.5 Parallelizing Critical Methods

Figure 5.10 presents the algorithm for parallelizing a critical method. First, it determines the number of new PE needed. For sequential and SRPC execution models, no new PE is needed. For ARPC model, a new PE is needed. Then, the algorithm assigns all the calls to the critical method to the new PE. For ARPC and

```
estimatingOverlappingTime($s, Task, mode$);
  begin
    case mode = SEQ or SRPC :
      return 0;
    case mode = ARPC :
      $pre$ = findPredecessor($s, GDG(Task)$);
      $suc$ = findSuccessor($s, GDG(Task)$);
    case mode = ARPC&CLONING :
      $pre$ = findPredecessor($s, SPG(Task)$);
      $suc$ = findSuccessor($s, SPG(Task)$);
    end case;
    /* calculate overlapping time */
    $overlap = 0$;
    $tmp = 0$;
    for each node $n$ between $s$ and $suc$ do
      $tmp = tmp + ET(n)$;
    end for;
    if $tmp > ET(s) + CC(s)$ then
      $overlap$ = tmp;
    else
      $overlap = ET(s) + CC(s)$;
    end if;
    return $overlap$;
  end;
```

Figure 5.9 Algorithm for estimating overlapping time.

ADT cloning, the upper bound number of new PEs are needed since the method is fully cloned up to the upper bound. Calls to the critical method is assigned to one of the new PEs in a round-ribbon since any of the new PEs contains a clone of the critical method. After a call to the critical method is reassigned, the old schedule is updated (both EST and LST are modified). The two nested loops contribute the complexity of this algorithm which is $O(S^2)$.

The overlapping times of critical methods are calculated (shown in Table 5.1). By examining the overlapping execution times of all the calls to the critical methods, the critical method with the largest overlapping execution time is chosen. Method *initialization* has the largest overlapping execution time if it is parallelized. Parallelization via ARPCs gains less concurrency but needs fewer resources.

Figure 5.11 shows the CFG of VM_Simu after reallocating method *initialization* to another processor. The execution time is reduced to a value between 4087 and

```
ParallelizingCriticalMethod($Task$,$CM$,$mode$)
  begin
      /* number of new PEs needed */
    case mode = SEQ or SRPC :
      $numNewPE = 0$;
    case mode = ARPC :
      $numNewPE = 1$;
    case mode = ARPC&CLONING :
      $numNewPE = CR(CM)$;
      end case;
    for each call $s$ to $CM$ in $Task$ do
      case mode = SEQ or SRPC :
        return 0;
      case mode = ARPC :
        $suc$ = findSuccessor($s, GDG(CM)$);
        schedule $s$ at time $EST(s)$ in PE $PE(numPEUsed + numNewPE)$;
      case mode = ARPC&CLONING :
        $suc$ = findSuccessor($s, SPG(CM)$);
        $lastUsedPE = (numPEUsed + 1)$ % $(numNewPE)$;
        schedule $s$ at time $EST(s)$ in PE $PE(lastUsedPE)$;
      end case;
      remove $s \rightarrow s+1$ from $CFG(Task)$;
      add $pre \rightarrow s$ into $CFG(Task)$;
      add $s \rightarrow suc$ into $CFG(Task)$;
      for each node $n$ between $s$ and $suc$ do
          /* modify the earliest execution time of statement $n$ */
        $EST(n) = EST(n) - EST(CM) + CC(CM)$;
          /* modify the latest execution time of statement $n$ */
        $LST(n) = LST(n) - LST(CM) + CC(CM)$;
      end for;
      for each successor node $n$ following $suc$ do
        $EST(n) = EST(n) - OET$;
        $LST(n) = LST(n) - OET$;
      end for;
    end for;
      /* $numPEUsed$ is a global variable */
    $numPEUsed = numPEUsed + numNewPE$;
  end;
```

Figure 5.10 Algorithm for parallelizing a critical method.

4452 time units. At least 396 time units are gained via ARPCs. If this still does not meet the deadline, ARPCs combined with method cloning are used. Figure 5.12 shows the CFG after making two clones of the method *initialization* and placing them to another two processors. The execution time can be reduced to 3877 time units.

If ADT instances are used as cloning and distribution units, Table 5.2 shows the precedence and the overlapping times of critical call statements. Figure 5.13 shows the CFG of VM_Simu after reallocating ADT instance *Processor* to another

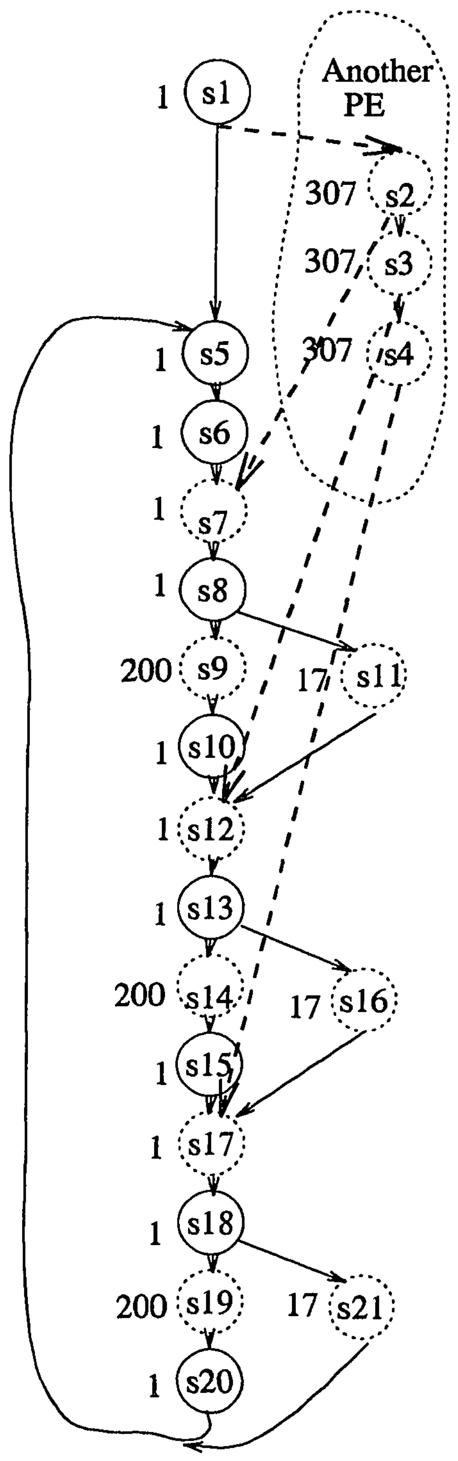


Figure 5.11 The CFG of procedure *VM_Simu* after reallocating *initialization*.

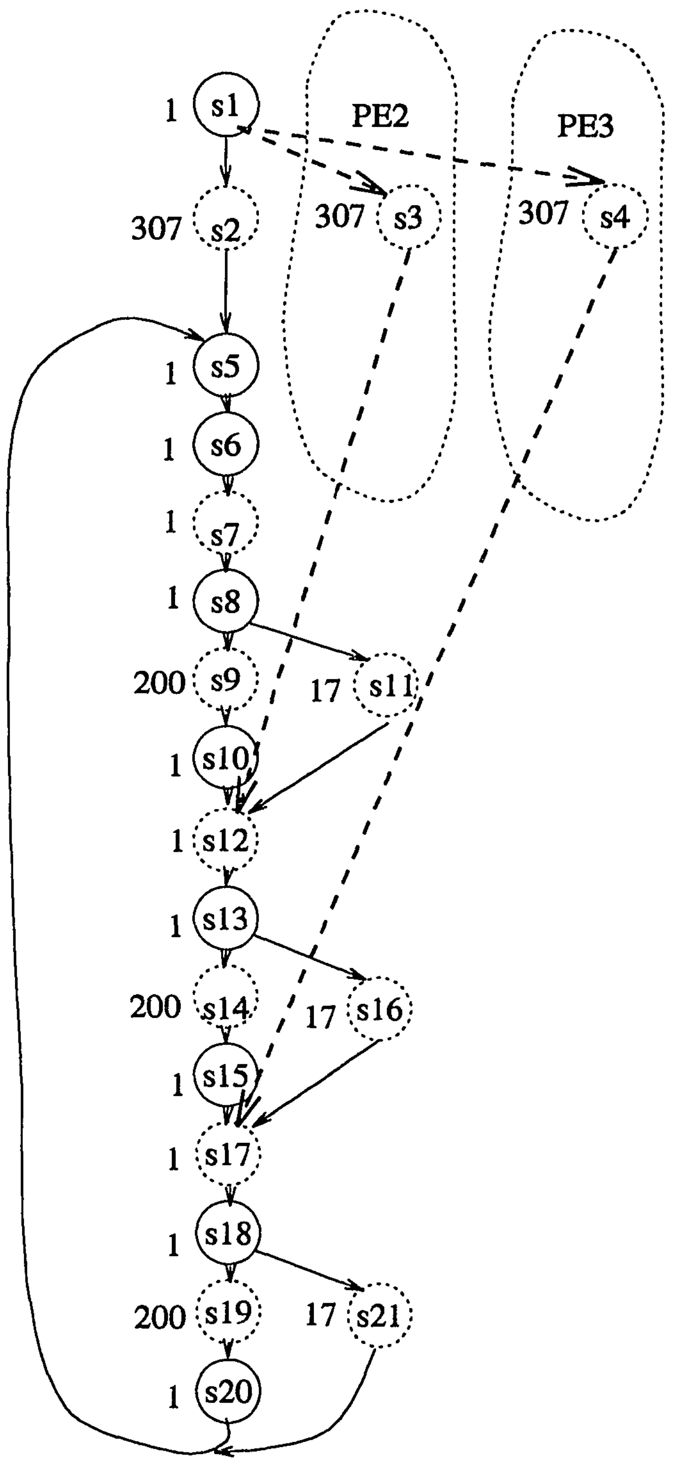


Figure 5.12 The CFG of procedure *VM_Simu* after cloning *initialization*.

Table 5.1 The critical methods of the schedule for procedure *VM_Simu*.

Called Method	*Nodes*	*via ARPC*			*via ARPC&Cloning*		
		predecessor	*successor*	*overlap ET*	*predecessor*	*successor*	*overlap ET*
initialization	S2		S3	1		S7	307
	S3	S2	S4	0		S12	307
	S4	S3	S17	40-307		S17	307
GetState	S7	S5	S8	1	S5	S8	1
	S12	S7	S13	1	S5	S13	1
	S17	S12	S18	1	S5	S18	1
dumpDataMemory	S9	S8	S14	3	S8	S14	3
	S14	S13	S19	3	S13	S19	3
	S19	S18		1	S18		1
doinstruction	S11	S8	S16	2	S8		17
	S16	S13	S21	2	S13		17
	S21	S18		1	S18		17

processor. The execution time is increased to 8024.5 time units! The negative speedup is caused by the communication overhead. Since some methods of the ADT instance *Processor* have very small execution times (method *GetState* takes 1 time unit), much more time is spent on communication than on execution of the methods. This is the disadvantage compared with method level cloning and distribution. ARPCs combined with ADT instance cloning gain the same speedup compared with the method level cloning. Figure 5.14 shows the CFG after making two clones of the instance *Processor* and placing them to another two processors. The execution time can be reduced to 3877 time units.

5.6 Complexity Analysis

The incremental scheduling algorithm (shown in Figure 5.1) calls *ConstructStatementTable* (a parser) to create statement tables. Scanning and parsing a program

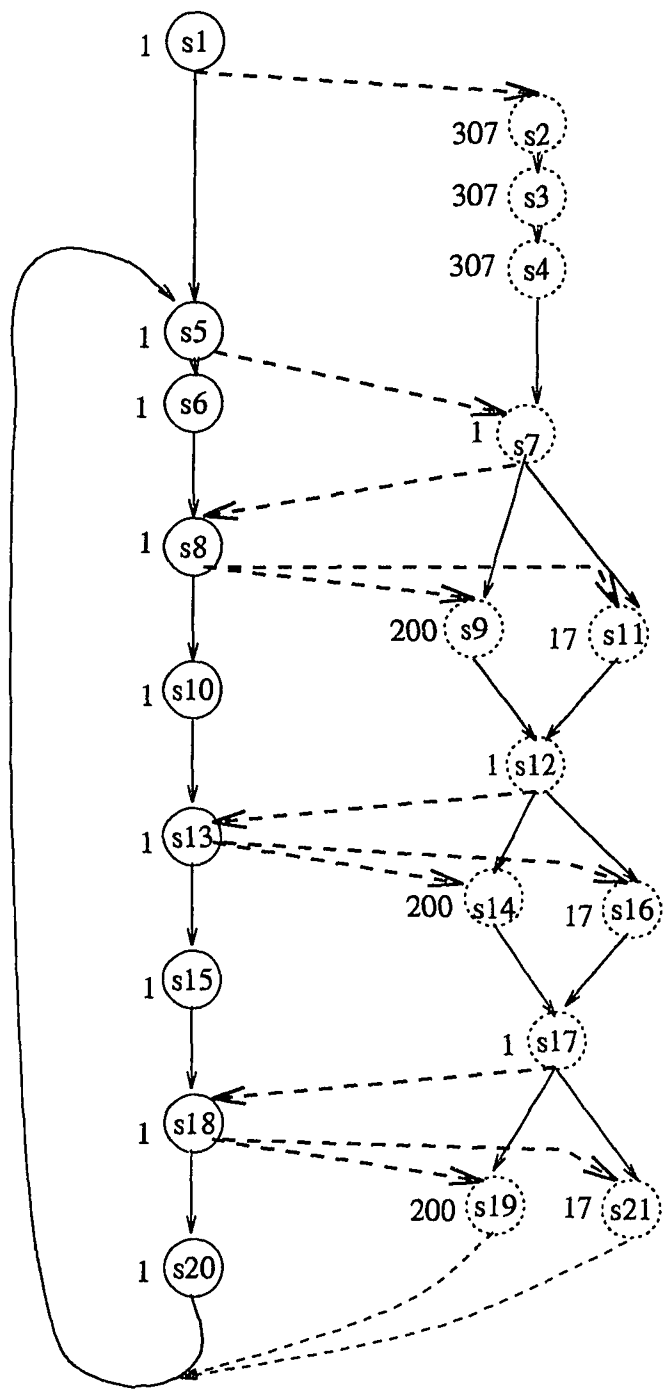


Figure 5.13 The CFG of procedure *VM_Simu* after reallocating *Processor*.

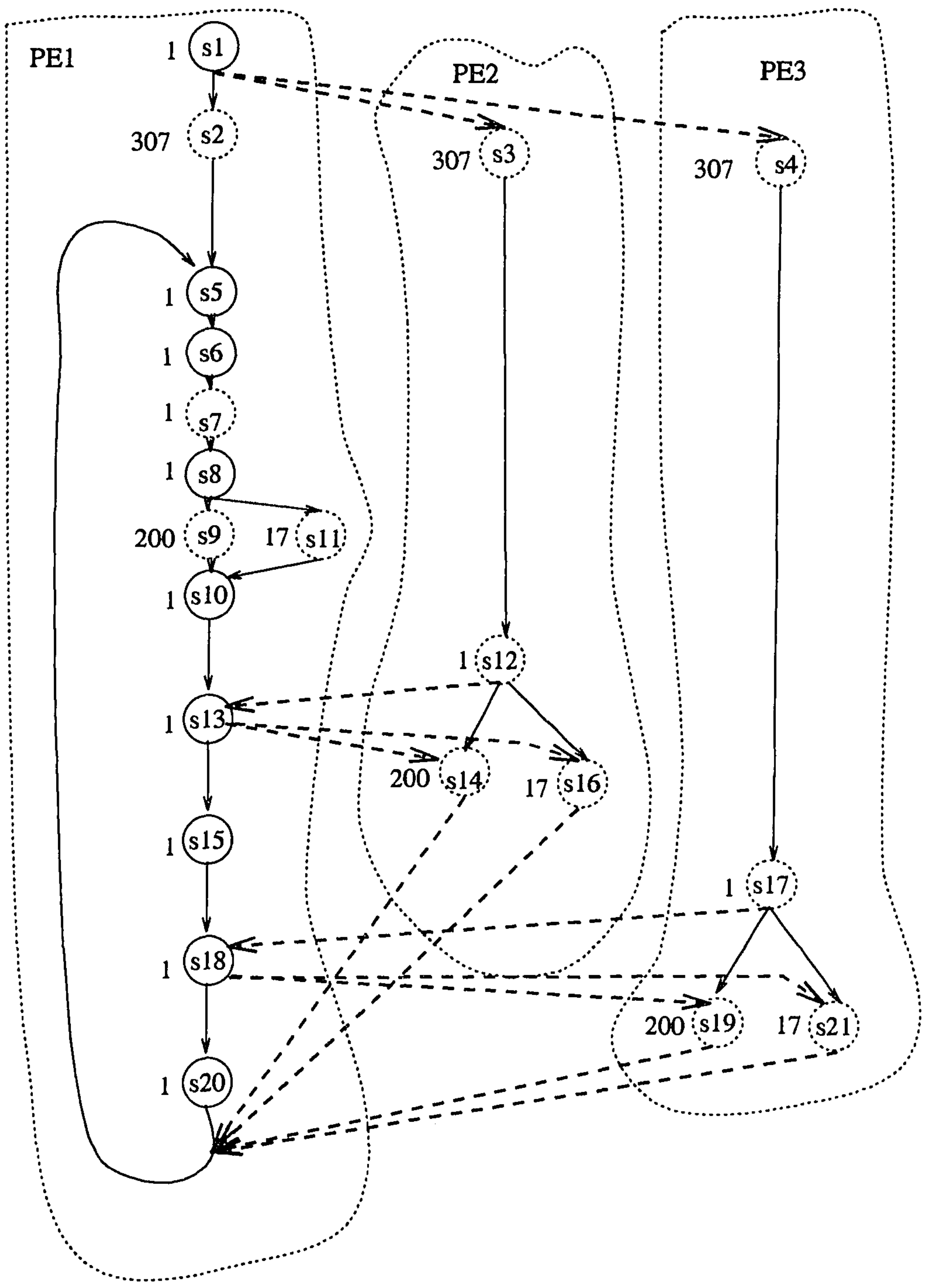


Figure 5.14 The CFG of procedure *VM_Simu* after cloning *Processor*.

Table 5.2 The critical instance of the schedule for procedure VM_Simu.

Called Instance	*Nodes*	*via ARPC*			*via ARPC&Cloning*		
		predecessor	*successor*	*overlap ET*	*predecessor*	*successor*	*overlap ET*
Processor	S2		S3	1		S7	307
	S3	S2	S4	0		S12	307
	S4	S3	S7	2		S17	307
	S7	S5	S8	1	S5	S8	1
	S12	S9,S11	S13	1	S5	S13	1
	S17	S14,S16	S18	1	S5	S18	1
	S9	S8	S12	1	S8	S14	3
	S14	S13	S17	1	S13	S19	3
	S19	S18		1	S18		1
	S11	S8	S16	2	S8		17
	S16	S13	S21	2	S13		17
	S21	S18		1	S18		17

one time are polynomial complexity which is $O(M \cdot S)$ (M is the number of methods in an application, and S is the maximum number of statements in a method).

As discussed in Section 4.2, constructing a method call graph for an application takes $O(M \cdot S)$.

As discussed in Chapter 3 and Chapter 4, the dependence and cloning analysis have also polynomial complexity.

The initial schedule construction takes $O(M \cdot S)$.

Dependence analysis takes $O(M \cdot S^3)$ to create all the dependence graphs (as discussed in Chapter 3.

Cloning analysis takes $O(M^3 \cdot S^4)$ to calculate CRs on method level.

Agreggating CRs from method level to ADT instance level takes $O(I \cdot S)$ I is the maximum number of methods in an ADT module.

Topological sorting takes $O(MlogM)$.

Identifying critical methods takes $O(S^2)$.

Estimating communication overhead takes $O(S^2)$.

Estimating overlapping time of a call statement takes $O(S)$.

Critical Method Parallelization takes $O(M \cdot S)$ to update the schedule.

Therefore, the complexity of the incremental scheduling algorithm is $O(M^3 \cdot S^4)$.

CHAPTER 6

EXPERIMENTAL RESULTS

To show how the concurrency enhancement techniques improve the execution performance of programs constructed with ADT instances, we look at both real application programs and a large number of random programs. A program for a virtual machine simulation is used to show the utility of the concurrency extraction techniques. Experiments are conducted to compare the speedup of parallelized programs against sequential programs. To get more test cases, a program generator was developed. In order to set meaningful parameters for the program generator, a survey of ADT-based programs from the public domain was conducted. Two hundred eighty-eight different programs were generated and assessed on six distributed configurations.

This chapter is organized as follows. The experimental approach is introduced first. This is followed by presenting the experimental results for the virtual machine simulation program. Section 6.3 describes the techniques used to generate realistic programs. In Section 6.4, the experimental results for the programs generated by the program generator are presented.

6.1 Experimental Approach

An overview of our experimental approach is shown in Figure 6.1. In the experiments, four execution modes are evaluated and compared. The four execution modes are:

1. sequential execution with one processor;
2. synchronous remote procedure calls (SRPCs) with multiple processors
3. asynchronous remote procedure calls (ARPCs) with multiple processors

4. ARPCs with cloning

The parameters of the experiment are:

- number of ADT instances,
- percentage of primitive ADT instances,
- maximum method size,
- minimum method size,
- percentage of call statements, and
- the ratio of number of precedence relations to the number of statements.

The experimental approach (shown in Figure 6.1) is as follows:

- (1). A method call graph (MCG) is constructed to describe the use relations among methods.
- (2). The information hiding (IH) metric is computed as the scaled average height of the MCG. It measures the layering of the methods. The larger the IH metric is, the more layering the system has. The formula for IH is:

  $$IH = \frac{H}{NP \times NN}$$

  Where H is the height of the MCG.
  NP is the number of paths from root to leaves in the MCG.
  NN is the number of nodes in the MCG.
- (3). A CFG is constructed for each method to describe the control flow of the method.
- (4). A sequential schedule of each method is constructed based on the CFG of the method. Since only one processor is used, the flow of control of each method forms a sequential schedule.

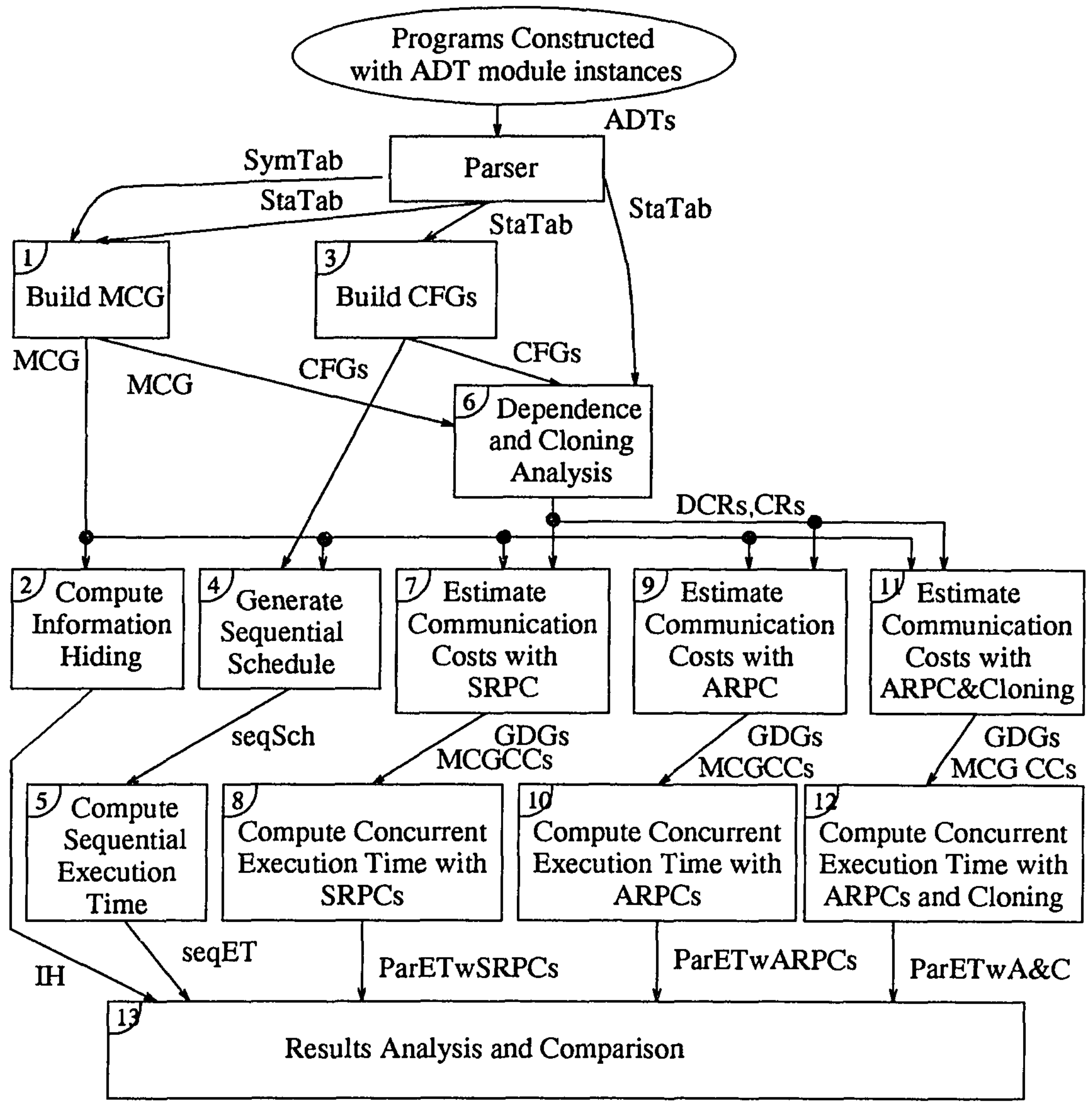

Programs Constructed with ADT module instances
ADTs
Parser
SymTab
StaTab
StaTab
StaTab
1 Build MCG
3 Build CFGs
MCG
MCG
CFGs
CFGs
6 Dependence and Cloning Analysis
DCRs,CRs
2 Compute Information Hiding
4 Generate Sequential Schedule
7 Estimate Communication Costs with SRPC
9 Estimate Communication Costs with ARPC
11 Estimate Communication Costs with ARPC&Cloning
seqSch
GDGs MCGCCs
GDGs MCGCCs
GDGs MCG CCs
5 Compute Sequential Execution Time
8 Compute Concurrent Execution Time with SRPCs
10 Compute Concurrent Execution Time with ARPCs
12 Compute Concurrent Execution Time with ARPCs and Cloning
IH
seqET
ParETwSRPCs
ParETwARPCs
ParETwA&C
13 Results Analysis and Comparison

Figure 6.1 Overview of the experimental approach.

- (5). The sequential execution time of a method is calculated using the algorithm presented in Figure 5.3.
- (6). CDGs, DDGs, and GDGs are constructed for each method. An MDG is also constructed for each used method. The DCR of each method is determined. An IMDG is constructed to calculate CR for each shared method. The total clone requirements (CRs) of each used ADT instance are determined by propagating the DCRs bottom-up in the MCG.
- (7). (9). (11). The contention for the communication is considered for each of the execution paradigms (see Section 5.3).
- (8). Although the execution of a method via SRPCs crosses multiple processors, it is sequential execution since the caller is blocked on each method call until the call returns. The CFG of the method is used to evaluate the execution time of a method via SRPCs since the CFG represents the sequential flow of control among the statements of methods. Using algorithm *calculateET* (shown in Figure 5.3) with the CFG as the parameter, the execution time of a method via SRPCs (without considering communication cost) is calculated first. The communication cost is subsequently added to the execution time of the method.
- (10) The execution time of a method with the ARPC paradigm can be evaluated using the GDG of the method. Using algorithm *calculateET* (shown in Figure 5.3) with the GDG as the parameter, the execution time of a method with ARPCs (without considering communication cost) is calculated first. The communication cost is added to this execution time.
- (12) Since each method is cloned up to the upper bound, all code contention is resolved, therefore, all the code dependence relations in the GDG of the method are removed; only data and control dependence relations must be obeyed. Thus,

using algorithm *calculateET* (shown in Figure 5.3) with SPG as the parameter, the execution time of a method with ARPCs and cloning paradigm is calculated first, and the communication cost is added to the execution time.

- (13). Speedup is measured as the ratio of sequential execution time to the execution time with each of SRPC paradigm, ARPC paradigm, and ARPC and cloning paradigm.

6.2 Virtual Machine Simulation

In this section, the virtual machine simulation program is used to demonstrate the concurrency extraction techniques. As shown in the MCG of the virtual machine simulation application (in Figure 4.3), there are seven primitive methods (*DataMem.initialize*, *DataMem.store*, *DataMem.fetch*, *InstrMem.initialize*, *InstrMem.store*, *InstrMem.fetch*, and *Processor.getState*), three synthesized methods (*Processor.initialization*, *Processor.doInstruction*, and *Processor.dumpDataMemory*), and one top level main procedure *VM_Simu*.

Assuming only one connection between any pair of processors, for each method in the virtual machine simulation, the communication costs (CCs) are estimated using the approach presented in Section 5.3 (examples are also presented in the section), and the execution times (ETs) are calculated using the algorithm presented in Figure 5.3. The ETs and CCs of methods in the virtual machine simulation are summarized in Table 6.2. The speedup is determined by calculating the ratio of the sequential execution time to the execution times via SRPCs, ARPCs, and ARPCs&Cloning, respectively. This is summarized in Table 6.2 for a 1-network. From the above results, we can see that the execution time via SRPCs is the slowest because it is a sequential execution, and the communication overhead makes it slower than sequential execution on one processor. Execution via ARPCs is faster than execution via SRPCs because of the concurrency gained, but it is slower than

Table 6.1 The execution times of methods in the virtual machine simulation with a 1-network.

ADT Instances	*Methods*	*SEQ*	*SRPC*		*ARPC*		*ARPC&Cloning*	
		ET	*CC*	*ET*	*CC*	*ET*	*CC*	*ET*
InstrMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
DataMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
Processor	Initialization	307	129	436	129	430	129	430
	doInstruction	17	5	22	5	22	5	22
	getState	1	0	1	0	1	0	1
	dumpDataMemory	200	125	325	125	325	125	325
	VM_Simu	4483	2047.5	8042.5	2047.5	8024.5	2753.5	4812.5

Table 6.2 The speedup of *VM_Simu* with a 1-network.

Task	*Speedup*		
	Seq/SRPCs	*Seq/ARPCs*	*Seq/ARPCs&Cloning*
VM_Simu	0.557	0.558	0.932

sequential execution on one processor due to the contention for communication (this will improve as the interconnection topology grows). Execution via ARPCs and cloning gains more concurrency than execution via only ARPCs because of resolving contention for shared methods, but ARPCs and cloning together cause more network contention. One conclusion is that when network is not fast enough, ARPCs and cloning may not speedup the execution because of the communication overhead they cause. The execution times of procedure *VM_Simu* via four different execution modes with four different networks are compared in Figure 6.2. When the number of links is greater than one, ARPCs and cloning cuts execution time significantly.

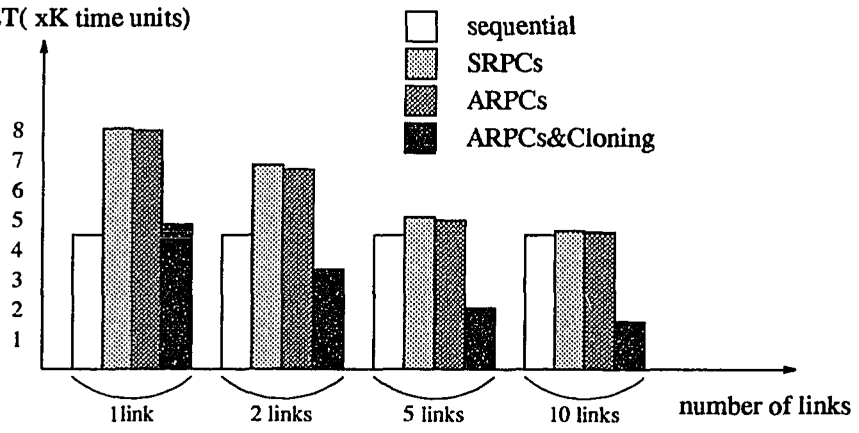


Figure 6.2 Execution times of VM_Simu in four different execution modes with four different networks.

When the number of connections among processors increases, the contention for communication decreases. Assume there are two connections between any pair of processors. The execution times via different execution modes are shown in Table 6.3. The speedups are presented in Table 6.4. Compared with one connection network, two connection network reduces the communication times, therefore, the speedup via different execution modes are increased. Note that with two connection network, the execution time of procedure VM_Simu via ARPCs and ADT cloning is shorter than sequential execution time. The execution times of procedure VM_Simu with SRPC paradigm and with ARPC paradigm are also reduced, but they are stilllonger than sequential execution time.

Assume there are five connections between any pair of processors. The execution times via different execution modes are shown in Table 6.5. The speedups is presented in Table 6.6. Note that with five connection network, the execution time of procedure VM_Simu via ARPCs and ADT cloning is half of the sequential execution time, i.e., procedure VM_Simu via ARPCs and ADT cloning twice faster than the sequential execution. The execution times of procedure VM_Simu with

Table 6.3 The execution times of methods in the virtual machine simulation with a 2-network.

Module	*Method*	*SEQ*	*SRPC*		*ARPC*		*ARPC&Cloning*	
Instances	Method	*ET*	*CC*	*ET*	*CC*	*ET*	*CC*	*ET*
InstrMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
DataMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
Processor	Initialization	307	96.8	403.8	96.8	397.8	96.8	397.8
	doInstruction	17	3.8	20.8	3.8	20.8	3.8	20.8
	getState	1	0	1	0	1	0	1
	dumpDataMemory	200	93.8	293.8	93.8	293.8	93.8	293.8
	VM_Simu	4483	1258	6882.8	1258	6863.8	1375	3310.6

Table 6.4 The speedup of *VM_Simu* with a 2-network.

Task	*Speedup*		
	Seq/SRPCs	*Seq/ARPCs*	*Seq/ARPCs&Cloning*
VM_Simu	0.651	0.653	1.354

Table 6.5 The execution times of methods in the virtual machine simulation with a 5-network.

Module Instances	*Method*	*SEQ*	*SRPC*		*ARPC*		*ARPC&Cloning*	
	Method	*ET*	*CC*	*ET*	*CC*	*ET*	*CC*	*ET*
InstrMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
DataMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
Processor	Initialization	307	40.8	347.8	40.8	341.8	40.8	341.8
	doInstruction	17	1.6	18.6	1.6	18.6	1.6	18.6
	getState	1	0	1	0	1	0	1
	dumpDataMemory	200	39.6	239.6	39.6	239.6	39.6	239.6
	VM_Simu	4483	241.3	5154.5	241.3	5136.5	313.9	2029.3

Table 6.6 The speedup of *VM_Simu* with a 5-network.

Task	*Speedup*		
	Seq/SRPCs	*Seq/ARPCs*	*Seq/ARPCs&Cloning*
VM_Simu	0.87	0.873	2.209

SRPC paradigm and ARPC paradigm are also reduced significantly, but they are still greater than sequential execution time.

Assume there are ten connections between any pair of processors. The execution times via different execution modes are shown in Table 6.7. The speedups is presented in Table 6.8. Note that with ten connection network, the execution time of procedure *VM_Simu* via ARPCs and ADT cloning is almost one third of the sequential execution time, i.e., procedure *VM_Simu* via ARPCs and ADT cloning three times faster than the sequential execution. This result is consistent with fact that three clones of the ADT instance *Processor* are made to resolve all

Table 6.7 The execution times of methods in the virtual machine simulation with a 10-network.

Module	*Method*	*SEQ*	*SRPC*		*ARPC*		*ARPC&Cloning*	
Instances	*Method*	*ET*	*CC*	*ET*	*CC*	*ET*	*CC*	*ET*
InstrMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
DataMem	Initialize	50	0	50	0	50	0	50
	fetch	1	0	1	0	1	0	1
	Store	1	0	1	0	1	0	1
Processor	Initialization	307	9.7	316.7	9.7	310.7	9.7	310.7
	doInstruction	17	0.4	17.4	0.4	17.4	0.4	17.4
	getState	1	0	1	0	1	0	1
	dumpDataMemory	200	9.4	209.4	9.4	209.4	9.4	209.4
	VM_Simu	4483	36.8	4637.1	36.8	4619.1	45.5	1639.6

Table 6.8 The speedup of *VM_Simu* with a 10-network.

Task	*Speedup*		
	Seq/SRPCs	*Seq/ARPCs*	*Seq/ARPCs&Cloning*
VM_Simu	0.967	0.971	2.734

the contention, therefore, ideally, the application should be three times faster. The execution times of procedure *VM_Simu* with SRPC paradigm and ARPC paradigm are also reduced significantly, now they are almost the same as sequential execution time.

6.3 Random Programs

In this section, an approach for generating random programs is introduced. A program generator is developed to generate random object-based programs. The parameters of the program generator are set according to the result of a real application survey. The program generator is introduced first. This is followed by the real application survey and parameter setting for the program generator.

6.3.1 Program Generator

A program generator was developed to test the concurrency extraction techniques. Since all the dependence and cloning analysis techniques are based on the intermediate representation (i.e., CFG, CDG, DDG, SPG, and GDG), it is unnecessary to generate code, instead, the generator generates the intermediate representation directly. The parameters of the generator (as shown in the Table 6.9) vary the programs generated in three levels of abstraction: task level, ADT instance level, and method level.

- *Number of tasks* is used to describe the number of top level processes. Processes are independent of each other. Typically, timing constraints (such as deadline, period, etc.) are defined at this level.
- *Number of ADT instances* is used to describe the size of the program generated.
- *Percentage of primitive ADT instances* is used to affect the layer of the program generated. The more primitive ADTs a program has, the fatter the instance call graph of the program.
- *Number of methods* is used to describe the size of an average ADT instance.
- *Number of parameters of call statements* is used to describe the size of the call and return packages.
- *Maximum number of statements* is used to describe the maximum number of statements which a method can have.
- *Minimum number of statements* is used to describe the minimum number of statements which a method can have.

Table 6.9 The parameters of the program generator.

Task level	Number of tasks
	Number of ADT instances
	Percentage of primitive ADT instances
Instance level	Number of methods
	Number of parameters of call statements
Method level	Maximum number of statements
	Minimum number of statements
	Percentage of call statements
	Ratio of number of data dependence edges to the number of nodes in the GDG

- *Percentage of call statements* is used to describe the amount of call statements in a method.
- *Ratio of number of data dependence edges to the number of nodes in the GDG* is used to describe the amount of data dependence edges in the GDG of a method.

The program generator generates a CFG for each method with the number of nodes in the range given as parameters. A certain percentage (given as parameters) of the statements of a method are generated as call statements. A CDG and a DDG of a method are generated based on the CFG of the method. To avoid cycles in the DDG, data dependence edges are generated so that the source node has a smaller label than the destination node. The same policy is used for generating call statements to avoid cycles in the ICG. By combining the CDG and DDG of a method, the SPG of the method is constructed. Adding the code dependence relations, the GDG of the method is built. With the CFGs and the GDGs of each method, the analysis techniques are applied. The information hiding and the speedup of a program are then calculated.

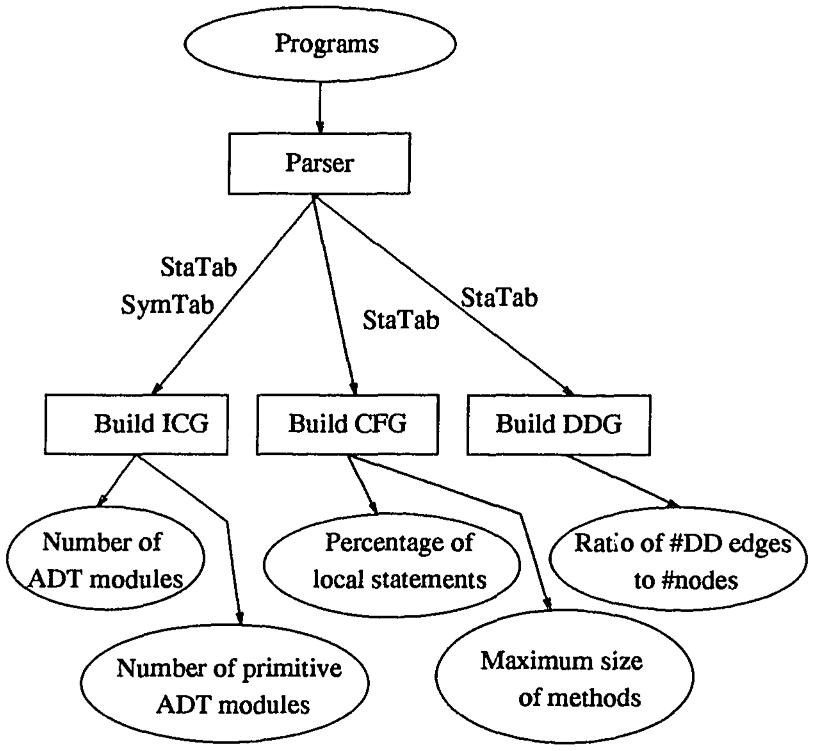


Figure 6.3 The flowchart of Ada dependence analysis tool set.

6.3.2 Application Survey and Parameters for the Program Generator

In order to get reasonable and meaningful parameters for the program generator, object-based applications from the public domain were studied (as shown in Table 6.10). The applications cover a wide range of areas such as simulation, performance monitoring, graph algorithms, sorting algorithms, and job scheduling algorithms. A tool set developed for dependence analysis of Ada programs [72], was used for collecting data from the Ada applications. The flowchart of the tool set is shown in Figure 6.3. The data collected for those applications is summarized in the Table 6.11.

Based on the survey results, the parameters of the generator are set to generate a large number of programs that are like real applications (as shown in Table 6.12).

Table 6.10 Ada Applications Collected From Public Domain.

System Name	Abbreviation	Description
System Status	SYS	Collects cpu performance data from each DEC station and intercepts ISIS group view changes to determine which functions are executing on which DEC station.
Elevator Simulation	ELR	Simulates the operation of an elevator
Heating Control Simulation	Heater	Simulates the automatic heater control system.
Spanning Forest	SPF	Implements the algorithm for finding a spanning forest for any input graph.
Job Scheduling	FMS	A job scheduling algorithm.

Table 6.11 Data Collected From the Ada Applications.

Parameters	*Applications*						
	SYS	*ELR*	*FMS*	*Heater*	*SPF*	*Total*	*Average*
Number of packages	41	21	22	8	21	113	23
Percentage of primitive packages	39	51	45	75	23	233	47
Maximum size of procedure	50	44	31	20	45	190	38
Percentage of local statements	52	71	87	81	40	330	66
(Number of DD edges)/(number of nodes)	1.17	0.41	0.34	0.62	1.36	3.90	0.78

Table 6.12 The parameters of the generator.

Parameters	*Parameter values*		
	low	mid	high
Number of ADT instances	25	50	100
Percentage of primitive ADT instances	25%		75%
Maximum method size	25		65
Minimum method size	20		60
Percentage of local statements	75%		90%
(Number of DD edges)/(number of nodes)	0.75		1.25

6.4 Results from Randomly Generated Programs

With seven network topologies (1, 2, 5, 10, 20, 50, ∞) and three runs each, 864 programs were generated and tested. Since the generator has six parameters, the 864 test results were grouped into six categories. Each category varies one of the six parameters. In this section, one from each category is discussed since cases in the same category have common features.

- Category 1: Each case in this category has parameters:
  - Number of ADT instances = 50.
  - Percentage of primitive ADT instances = 75%.
  - Maximum method size = 25.
  - Minimum method size = 20.
  - Percentage of local statements = 90%.
  - (Number of DD edges)/(number of nodes) = 0.75.

  With the above parameters, the generator was run three times. Since the programs are randomly generated, each run produces a different result (different programs and speedup). Figure 6.4 shows the speedups with ARPCs

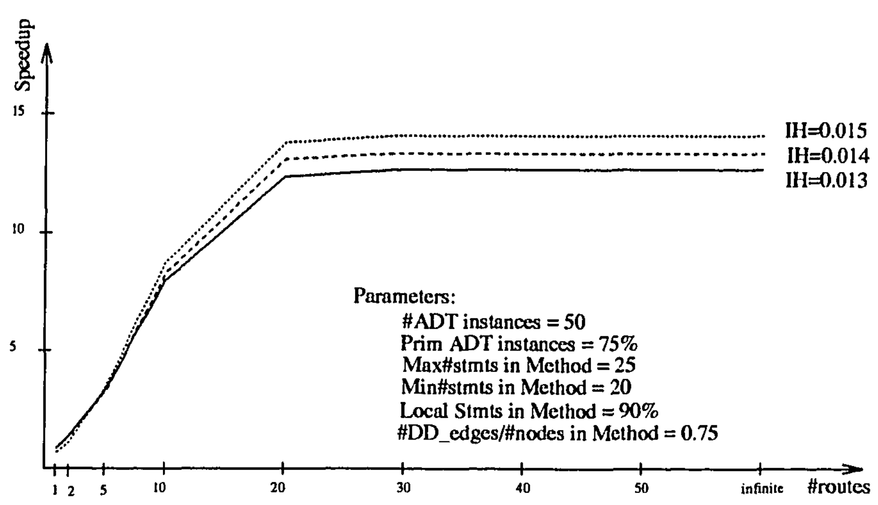


Figure 6.4 The experimental results when information hiding varies.

and cloning for the three cases. The figure shows that when IH increases, the speedup increases too.

Also, when the number of links between processors increases to 20 (or greater), the best speedup is achieved. Note that the figure shows the opposite results when the number of links is less than three. This is because the program with higher IH metric has more contention for the network than the program with lower IH metric when the number of communication links is few.

- Category 2: Each case in this category has parameters:
  - Number of ADT instances = variable $\in$ (25, 50, 100).
  - Percentage of primitive ADT instances = 75%.
  - Maximum method size = 25.
  - Minimum method size = 20.
  - Percentage of local statements = 90%.
  - (Number of DD edges)/(number of nodes) = 1.25.

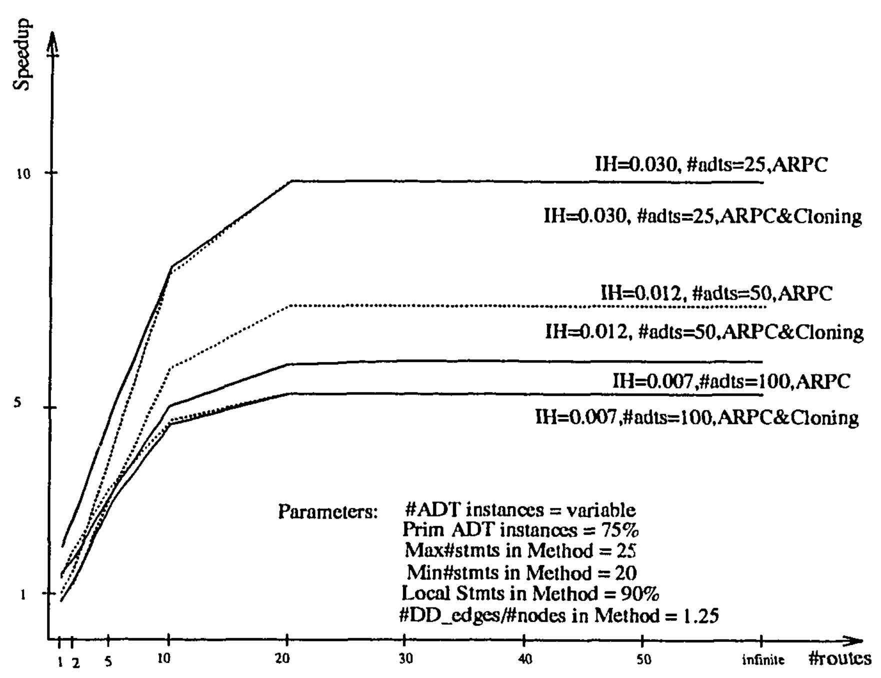


Figure 6.5 The experimental results when the number of ADT instances varies.

Figure 6.5 shows that when the size of programs increases, the speedup decreases. This is because the ICG becomes fatter (the IH metric decreases) when the number of ADT instance increases with all other parameters fixed. Note that the execution time with ARPC paradigm is better than the execution time with ARPC and ADT cloning paradigm when small (less than ten) network topology is used, and the execution time with ARPC paradigm and execution time with ARPC and ADT cloning paradigm are same when ten network topology is used. This shows that ARPC and ADT cloning paradigm may not lead to better programs due to the communication overhead.

- Category 3: One of the case in this category has parameters:
  - Number of ADT instances = 50.
  - Percentage of primitive ADT instances = variable ∈ (25%, 75%).
  - Maximum method size = 25.

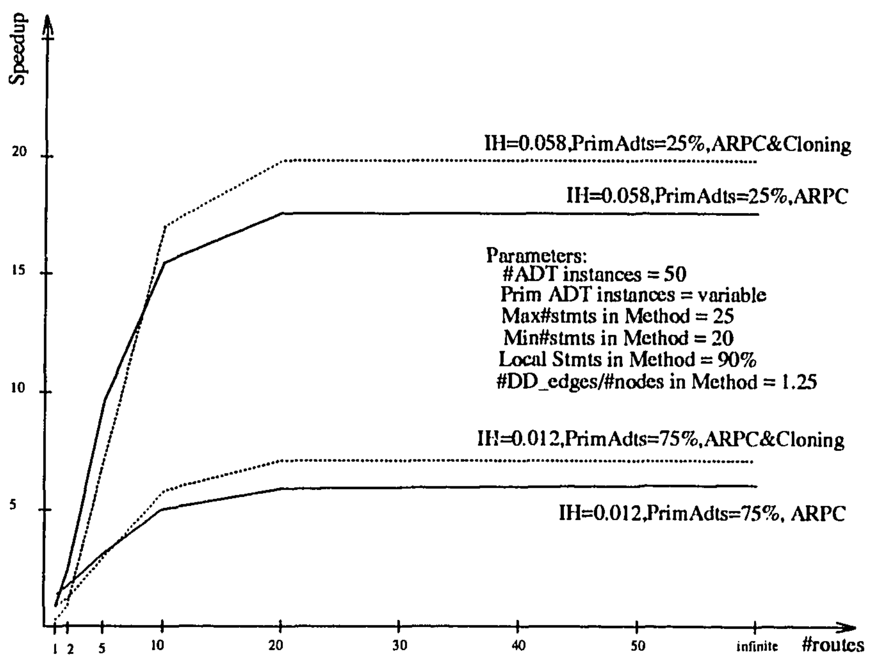


Figure 6.6 The experimental results when the percentage of primitive ADT instances varies.

- Minimum method size = 20.
- Percentage of local statements = 90%.
- (Number of DD edges)/(number of nodes) = 1.25.

Figure 6.6 shows that when the number of leaves of the ICG increases, the speedup decreases. This is because the ICG becomes fatter (the IH metric decreases) when the number of leaves of the ICG increases with all other parameters fixed.

- Category 4: One of the case in this category has parameters:
  - Number of ADT instances = 25.
  - Percentage of primitive ADT instances = 75%.
  - Maximum method size = variable ∈ (25,65).

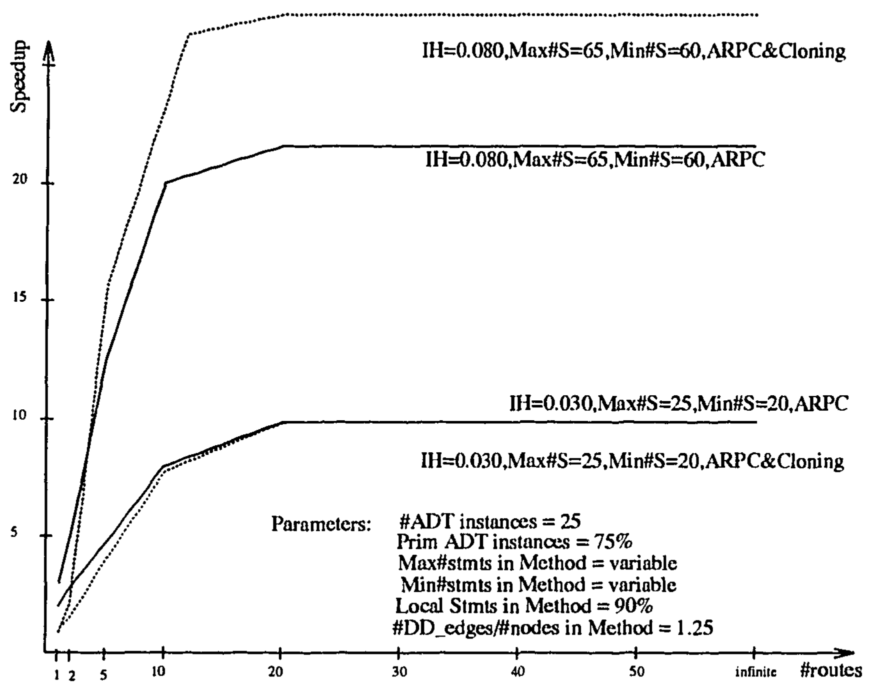


Figure 6.7 The experimental results when the number of statements in a method varies.

- Minimum method size = variable ∈ (20,60).
- Percentage of local statements = 90%.
- (Number of DD edges)/(number of nodes) = 1.25.

Figure 6.7 shows that The speedup increases when the size of method increases. This is because when the number of statements in a method increases with all other parameters fixed, the number of calls increases, thus, the chance for ARPC and cloning also increases.

- Category 5: One of the case in this category has parameters:
  - Number of ADT instances = 25.
  - Percentage of primitive ADT instances = 75%.
  - Maximum method size = 25.

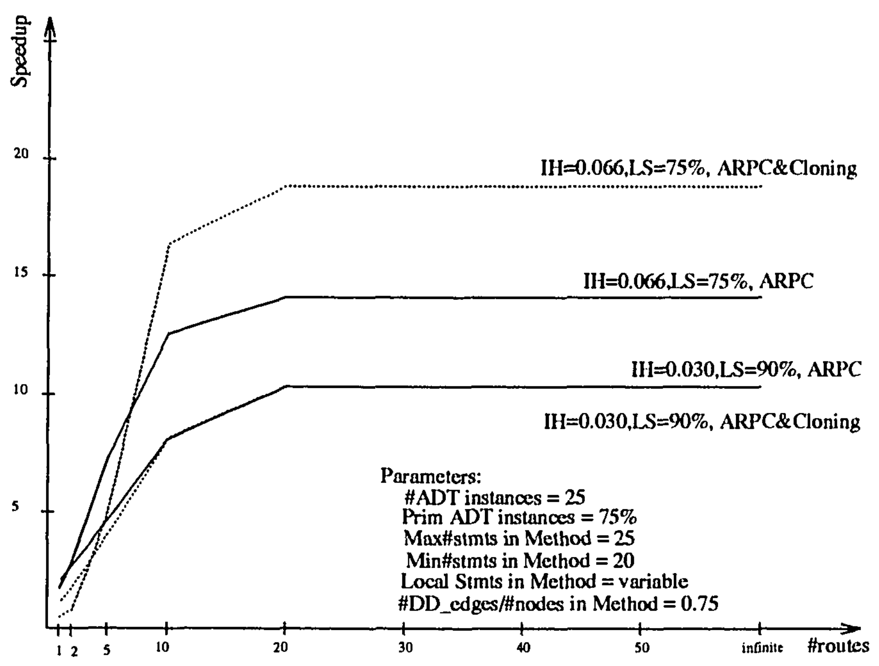


Figure 6.8 The experimental results when the percentage of local statements in a method varies.

- Minimum method size = 20.
- Percentage of local statements = variable ∈ (75%, 90%).
- (Number of DD edges)/(number of nodes) = 0.75.

Figure 6.8 shows the speedup decreases when the percentage of local statements in a method increases, because the number of calls in a method decreases.

- Category 6: One of the case in this category has parameters:
  - Number of ADT instances = 50.
  - Percentage of primitive ADT instances = 75%.
  - Maximum method size = 25.
  - Minimum method size = 20.
  - Percentage of local statements = 75%.

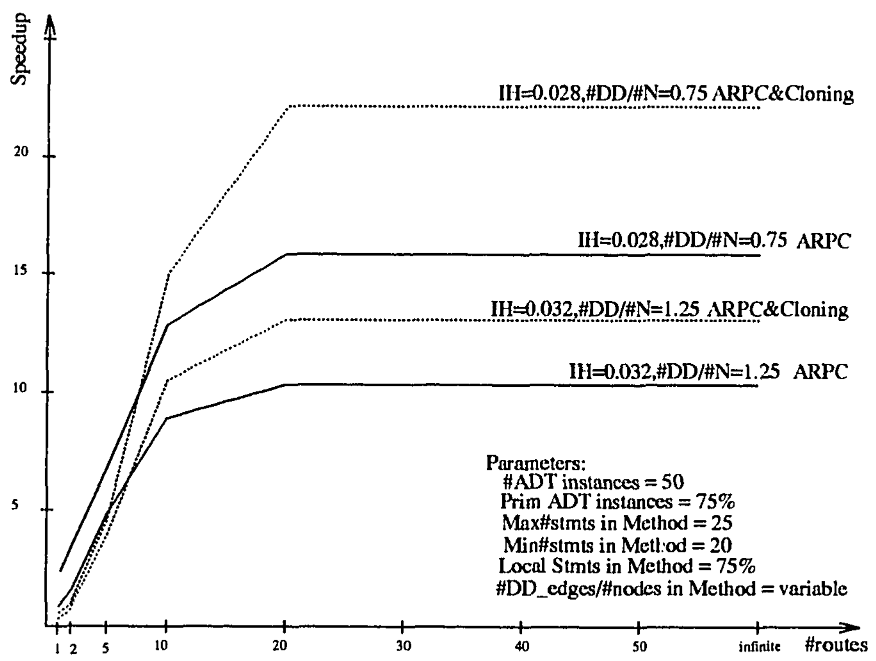


Figure 6.9 The experimental results when the data dependence ratio of a method varies.

- (Number of DD edges)/(number of nodes) = variable $\in$ (0.75, 1.25).

Figure 6.9 shows that the speedup decreases when the ratio of the number of data dependence in a method increases. This is because the number of calls that can be executed concurrently decreases when the the number of data dependence among statements increases with all other parameters fixed.

Figures 6.5 through 6.9 show that when the number of links between each pair of PEs is small (less than three), the speedup of execution with ARPCs is better than with ARPCs and cloning. This is because the clones of ADT instances add more contention to the network, and the communication contention delays remote procedure calls. From the above experimental results, we can get the following observations:

1. For object-based systems, the more layering the system has (the higher the IH metric), the more concurrency can be extracted.
2. The speedup decreases when the number of ADT instance increases, while all other parameters remain constant.
3. The speedup decreases when the percentage of the primitive ADT instance increases, but all other characteristics remain the same.
4. The speedup increases when the granularity of methods increases.
5. The speedup decreases when the percentage of local statements in a method increases.
6. The speedup decreases when the data dependence ratio of a method increases.
7. When the number of links between PEs is small, execution through ARPCs is better than through ARPCs combined with cloning.
8. ARPCs and ADT cloning can jield speedup of more than 20.

6.5 Experimental Assessment of Scheduling Techniques

To assess the incremental scheduling techniques, the scheduling techniques are evaluated for how often can a feasible schedule is found with incremental application of ARPCs and cloning. As shown in Figure 6.10, the deadline of a task is set to be a certain percentage (from 95 to 15) of its sequential execution time, then the analysis techniques are used to identify the critical paths and critical methods. The communication cost of each critical method is estimated. The critical method that has the largest ratio of execution overlap to communication overhead is chosen for parallelization. If the schedule is still infeasible, another iteration of identifying critical methods and parallelizing critical methods is performed until no more chances can

Table 6.13 The results of the experiment.

Sequential ET (SET)			*Deadline (D)*			*D/ SET*	number of iterations			*number of PE used*			FS	PFS
Min	Max	Avg	Min	Max	Avg		Min	Max	Avg	Min	Max	Avg		
742	1744958	224736	704	1657710	214545	95	1	24	5	2	25	6	33	94
742	1744958	224736	667	1570462	203254	90	2	36	7	3	37	8	31	89
742	1744958	224736	630	1483214	191962	85	3	51	10	4	52	11	31	89
742	1744958	224736	593	1395966	180670	80	4	66	12	5	67	13	27	77
742	1744958	224736	556	1308718	169378	75	5	61	13	6	62	14	27	77
742	1744958	224736	519	1221470	158086	70	6	96	17	7	97	18	27	77
742	1744958	224736	482	1134222	138393	65	7	101	20	8	102	21	27	77
742	1744958	224736	445	1046974	127747	60	8	116	22	9	117	23	26	74
742	1744958	224736	408	959726	117102	55	9	131	26	10	132	27	22	63
742	1744958	224736	370	872478	106456	50	10	146	29	11	147	30	17	49
742	1744958	224736	333	785230	88355	45	11	161	33	12	162	34	13	37
742	1744958	224736	296	697982	78538	40	12	176	36	13	177	37	9	26
742	1744958	224736	259	610735	68721	35	13	191	40	14	192	41	5	14
742	1744958	224736	222	523487	58903	30	14	206	44	15	207	45	4	11
742	1744958	224736	185	728811	59731	25	15	221	48	16	222	48	3	9
742	1744958	224736	148	191723	28302	20	16	236	52	17	237	53	2	6
742	1744958	224736	111	143792	21226	15	17	251	56	18	252	57	0	0

be found for speedup through concurrency exploitation. If a feasible schedule is found, the deadline of the task is reduced by another 5 percent. In this experiment, a 2-network is used for communication. Parallelization is applied at task level only in this experiment. Thirty-five random programs are examined.

Parameters in Table 6.12 are used to generate programs. Like the experiment for speedup, each time, one of the parameters is varied. The result of the experiments are shown in Table 6.13 (where D denotes deadline, ET denotes execution time, SET denotes sequential execution time, FS denotes feasible schedules found, and PFS denotes percentage of feasible schedules found). In this experiment, when the deadline of a task is set to be 95 percentage of its sequential execution time, 33 of 35 programs generated end with a feasible schedule. When the percentage is reduced to 85, the techniques fails to find feasible schedule for only four cases. The success rate is 89 percent. When the percentage is reduced to 65, 27 schedules are made feasible with an average of 21 PEs used. When the ratio of deadline to sequential execution increases, the chance for finding feasible schedules increases (as shown in Figure 6.11), the number of PEs required decreases (as shown in Figure 6.12), and the number of iterations performed decrease(as shown in Figure 6.13). One observation from the experimental results is that the deadlines of tasks should be set in the range of 65 to 95 percent of sequential execution execution time in order to get around 80 percent of feasible schedules with around 100 iterations.

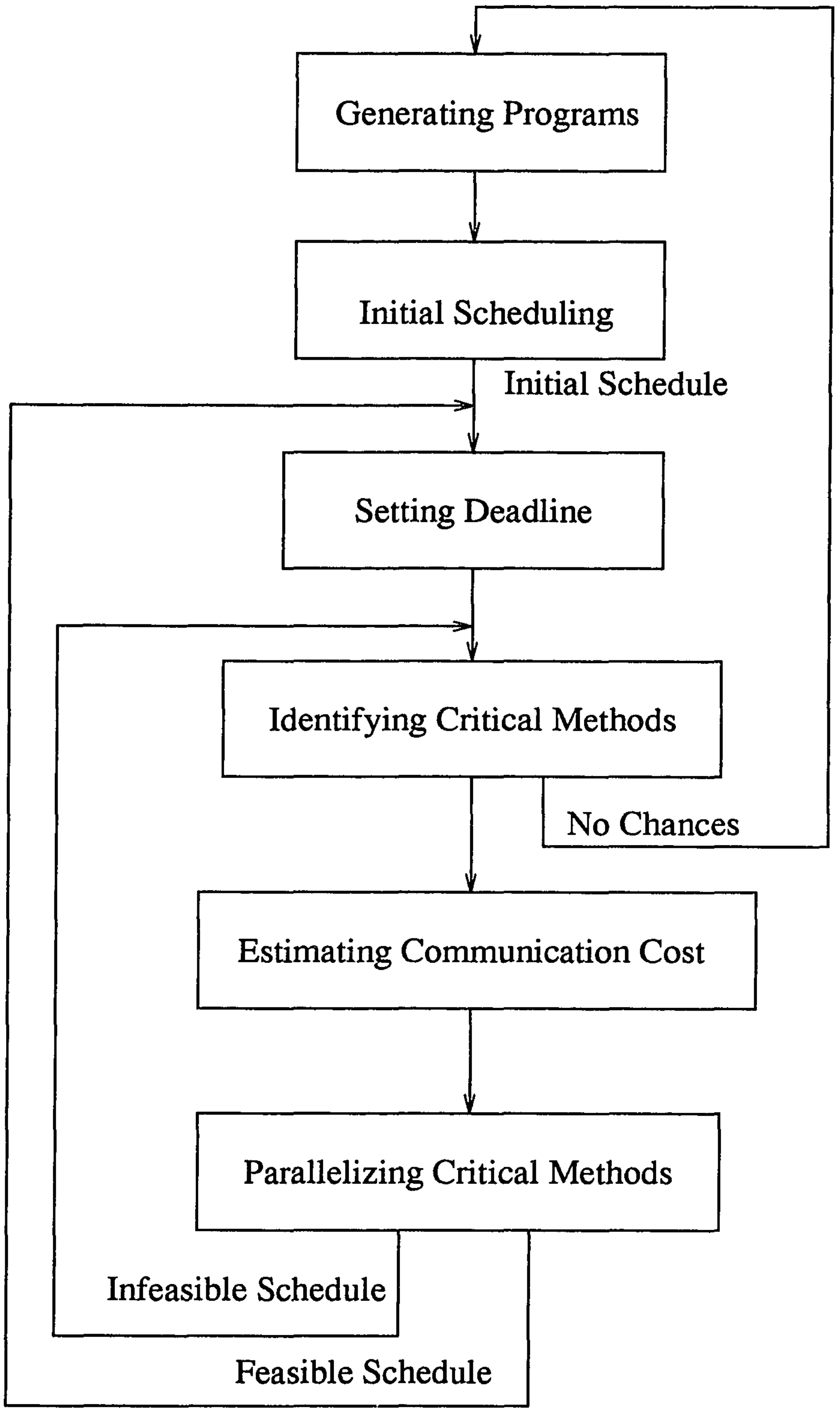


Figure 6.10 The flowchart of the experimental assessment.

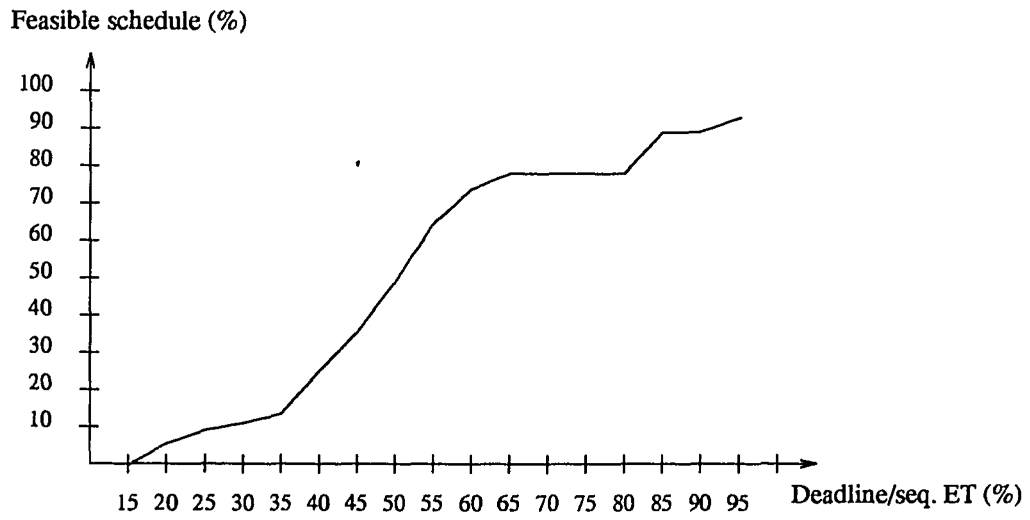


Figure 6.11 The relation between the ratio of deadline to sequential execution time and the percentage of finding feasible schedules.

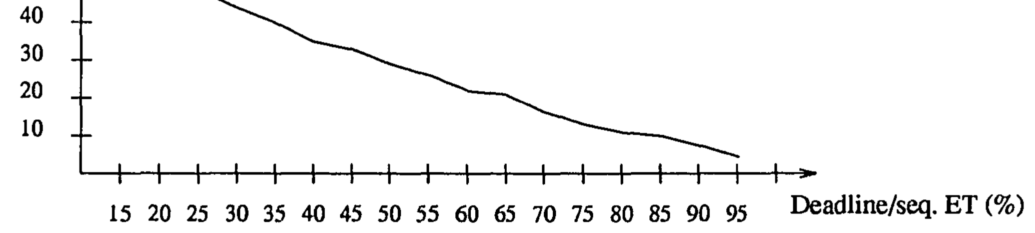


Figure 6.12 The relation between the ratio of deadline to sequential execution time and the number of PEs required.

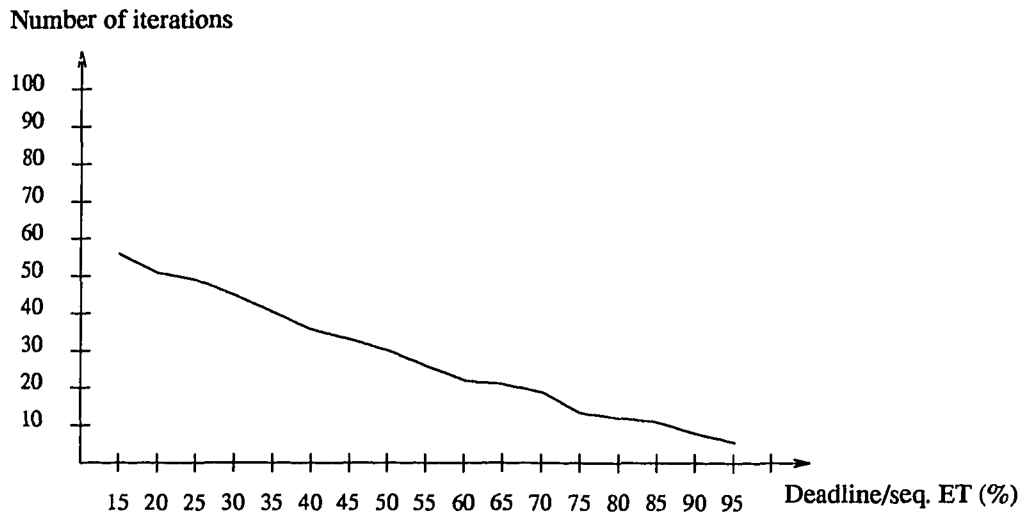


Figure 6.13 The relation between the ratio of deadline to sequential execution time and the number of iterations performed.

CHAPTER 7

CONCLUSION

Use of ADT modules can increase reusability of software components, but potential inefficiencies may occur at execution time due to the large number of procedure calls, and due to contention for shared ADTs in concurrent systems. Thus an ADT often becomes a bottleneck when those objects are manipulated simultaneously. On the other hand, the experimental results show that the potential for concurrency in ADT-based systems is quite high via the asynchronous remote procedure call (ARPC) and replication (cloning) of ADTs. Therefore, concurrent execution via ARPCs and ADT cloning can greatly improve the performance of programs.

This thesis presents a set of techniques for automatically identifying and exploiting concurrency in object-based systems at three levels of granularities: statement level, method level, and instance level. The program dependence graph, which was previously for describing data and control dependence, is extended to include code dependence relations. Code dependence is used to describe contention for ADTs. The general dependence graphs describe data, control, and code dependence relations. Properties of the general dependence graphs are formalized as theorems. Polynomial algorithms for constructing all the graphical representations are provided. With the general dependence graphs, cloning analysis techniques are developed to determine the upper bounds on the number of clones of ADT methods or instances that can run concurrently. The upper bounds are used to guide the incremental parallelization process for constructing off-line schedules in hard real-time systems. Real-time scheduling [76, 77] can be done in conjunction with concurrency enhancement to improve the timing behavior of processes missing deadlines, greatly increasing the chances for constructing a feasible schedule.

The concurrency extraction techniques presented in this work are valuable during the reengineering process. The dependence and concurrency analysis techniques are used to help the U.S. Navy's reengineering efforts for mission critical systems, such as the U.S. Navy's AEGIS system [52, 53]. Reverse engineering involves not only capturing the intermediate representation to correctly and easily understand the current system, but also analyzing the current system to identify potential concurrency. During the computer-based systems reengineering process, it is necessary to identify potential concurrency during reverse engineering so that the performance of the current system can be improved to meet the new requirements. The concurrency information is in the form of metrics that are used to guide the reengineering processes of software transformation and system configuration, which seek to produce a system with a high degree of concurrency. The metrics are also used to assess the concurrency in a reengineered system. The concurrency extraction techniques used for the reengineering process can be found in [72, 71].

Experimental results show that the concurrency extraction techniques can greatly improve the performance of programs constructed with ADTs. One general conclusion is that the more layers a system has, the more potential concurrency exists in the system.

The assumptions of the techniques presented in these thesis are:

- To enable efficient analysis, aliasing is not available to users.
- Cloning is applied only to stateless ADTs.
- The parallelization techniques are applied to single task with a deadline.
- As the experimental results show, a fast communication network is expected in order to handle the extra overhead and contention caused by concurrent execution of multiple clones of ADTs.

Future research include:

- Cloning of ADTs with state needs additional effort on maintaining state consistency(i.e., a protocol is needed).
- Since the scheduling algorithm works on critical methods at each incremental cycle, full cloning analysis is not necessary. A "lazy" cloning analysis (do clonability analysis on critical methods only) is needed to reduce the time complexity.
- A general scheduling algorithm is needed for handling multiple tasks with addtional timing constraints (release time, period, precedence relation among tasks).
- Currently, the parallelization techniques are applied at task level only, an extension to other levels (instance level and method level) needs to be done to exploit more concurrency.
- Heuristics for selecting which critical method to parallelize are needed.
- Currently, when parallelizing a method with ADT cloning, a critical method is fully parallelized up to the upper bound of number of clones. An improvement can be done by adding one clone of an ADT at one iteration of parallelization. This can slow down the speed of parallelization and save resources (PEs) to find a feasible schedule that needs the least amount of resources.

GLOSSARY

ADT	Abstract data type
ARPC	Asynchronous remote procedure call
CC	Communication cost
CDG	Control dependence graph
CFG	Control flow graph
CR	Clone requirement
CCS	Calling sequence set
D	Deadline
DCR	Direct clone requirement
DDG	Data dependence graph
DUMS	Directly used method set
ET	Execution time
EST	earliest start time
EPC	External procedure call
FT	finish time
GDG	General dependence graph
ICG	Instance call graph
IH	Information hiding

IMDG	Inter-method code dependence graph
IPC	Internal procedure call
LPC	Local procedure call
LST	Latest start time
MDG	Method dependence graph
MCG	Method call graph
OD	Out degree
PDG	Program dependence graph
PE	Processing element
RPC	Remote procedure call
SPG	Statement precedence graph
SRPC	Synchronous remote procedure call
TCR	Transitive clone requirement
TUMS	Transitively used method set
UM	Used method
UMS	Used method set
WOD	Weighted out degree
front	A function that returns the first item on the list
α	A factor that scales the parameter size and packet handling delay
β	A factor that scales number of concurrent ARPCs

REFERENCES

1. J.R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, pages 233–246. ACM SIGPLAN Notices 19(6), June 1984.

2. R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 241–249. ACM SIGPLAN Notices (23(7), July 1988.

3. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

4. R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271. ACM, June 1990.

5. U. Banerjee, S.C Chen, D.J. Kuck, and R.A. Towle. Time and parallel processor bounds for fortran-like loops. *IEEE Transactions on Computers*, C-28(9):660–670, September 1979.

6. U. Banerjee and D. Gajski. Fast evaluation of loops with if statement. *IEEE Transactions on Computers*, C-33(11):1030–1033, November 1979.

7. J.P. Banning. An efficient way to find the side-effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Symposium on Principles of Programming Languages*, pages 29–41. ACM Press, San Antonio, Texas, January 1979.

8. A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966.

9. M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 162–175. ACM SIGPLAN Notices 21, 7, July 1986.

10. D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 152–161. ACM SIGPLAN Notices 21(7), July 1986.

11. V. Cherkassky. Redundant task-allocation in multicomputer systems. *IEEE Transactions on Software Engineering*, 41(3):336–342, September 1992.

12. J.D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced alias and side effects. In *Conference Record of the 20th Symposium on Principles of Programming Languages*, pages 232–245. ACM Press, Charleston, S. Carolina, January 1993.

13. J.D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitions program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, February 1994.

14. E. C. Cooper. Circus: A replicated procedure call facility. In *Proceedings of 4th Symposium on Reliability in Distributed Software and Databases*, pages 11–24, 1984.

15. K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. In *The International Conference on Computer Languages.* IEEE, April 1992.

16. K.D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the 16th Symposium on Principles of Programming Languages*, pages 49–59. ACM Press, Austin, Texas, January 1979.

17. R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

18. K. Driscoll and K. Hoyme. The airplane information management system: An integrated real-time flight-deck control system. In *Real-Time Systems Symposium.* IEEE, 1992.

19. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transaction on Programming Languages and Systems*, 9(3):319–349, July 1987.

20. Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting A Compiler.* Benjamin/Cummings, 1988.

21. R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Real-Time Systems Symposium*, pages 232–242. IEEE, December 1993.

22. R. Gerber and S. Hong. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering*, 21(5):380–404, May 1995.

23. P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems. In *Real-Time Systems Symposium*, pages 247–256. IEEE, December 1990.

24. D.K. Hammer and O.S. van Roosmalen. An object-oriented model for the construction of dependable distributed systems. In *International Workshop on Object-Orientation in Operating Systems (I-WOOOS 92)*, Paris, France, September 1992.

25. D. Harms and B.W. Weide. Types, copying, and swapping: Their influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

26. W.L. Harrison and D.A. Padua. Representing s-experisons for the efficient evaluation of lisp on parallel processors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 703–710. IEEE, August 1986.

27. M. J. Harrold, B. A. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. Technical Report 92-128, Clemson University, December 1992.

28. Joseph E. Hollingsworth. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada.* PhD thesis, The Ohio State University, December 1992.

29. S. Hong and R. Gerber. Compiling real-time programs into schedulable code. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28(6), pages 166–176. ACM, June 1993.

30. H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE MICRO*, 9(1):25–40, February 1989.

31. D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolf. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Symposium on Principles of Programming Languages*, pages 207–218. ACM, January 1981.

32. D.J. Kuck, Y. Muraoka, and S.C. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

33. L. Lamport. The parallel execution of do loops. *Communication of the ACM*, 17(2):83–93, February 1986.

34. W. Landi, B.G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, volume 28 (6), pages 56–67. ACM SIGPLAN Notices, (Albuquerque, New Mexico, June) 1993.

35. B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 260–267. ACM, June 1988.

36. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, January 1973.

37. C. Martel. Preemptive scheduling with release times, deadlines, and due times. *Journal of ACM*, 29(3):812–829, July 1982.

38. S.P. Midkiff and D.A. Padua. Compiler generated synchronization for do loops. In *Proceedings of the 1986 International Conference on Parallel Processing*. IEEE, August 1986.

39. A. Mok and M. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the 7th Texas Conference on Computing Systems*, pages 5.1–5.12, November 1978.

40. Department of Defense. *Reference Manual for the Ada 95*. Ada Joint Program Office, Washington, D.C., Government Printing Office, ansi/mil-std-1815a-1983 edition, 1983.

41. D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communication of the ACM*, 29(12):1184–1201, December 1986.

42. D.A. Padua, D.J. Kuck, and D.H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):889–895, September 1980.

43. Phil Pfeiffer. *Dependence-Based Representations for Programs with Reference Variables*. PhD thesis, University of Wisconsin-Madison, August 1991.

44. K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *International Conference on Distributed Computing Systems*. IEEE, May-June 1990.

45. K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. In *Proceedings of the 4th IEEE International Conference on Distributed Computing Systems*, pages 96–107, May 1984.

46. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report Technical Report CMU-CS-87-181, Carnegie-Mellon University, November 1987.

47. T. Shepard and J. A. M. Gagne. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software Engineering*, 17(7):669–677, July 1991.

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

49. B. Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal of Computing*, 12:294–299, May 1983.

50. M. Sitaraman, L. R. Welch, and D. E. Harms. Influences of a component-based industry on the expression of specifications of reusable software. *The International Journal of Software Engineering and Knowledge Engineering*, pages 207–229, June 1993.

51. J.A. Stankovic. Misconceptions about real-time computing. *IEEE Transactions on Computers*, 21(10):10–19, October 1988.

52. K. J. Stein. Aegis fleet defense nearing sea test. *Aviation Week and Space Technology*, 13(8):32–35, August 1973.

53. K. J. Stein. Aegis system tested successfully. *Aviation Week and Space Technology*, 7(4):36–40, April 1975.

54. A.D. Stoyenko, L.R. Welch, P.L. Laplante, T.J. Marlowe, C. Amaro, B. Cheng, A. K. Ganesh, M. Harelick, X. Jin, M. Younis, and G. Yu. A platform for complex real-time applications. In *The Complex Systems Engineering Synthesis and Assessment Technology Workshop*. Naval Surface Warfare Center, July 1993.

55. G.S. Tjaden and M.J. Flynn. Detection and parallel execution of independent interactions. *IEEE Transactions on Computers*, C-19(10):889–895, October 1970.

56. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 176–185. ACM SIGPLAN Notices 21, 7, July 1986.

57. J. D. Ullman. NP-complete scheduling problems. *Journal of Computing System Science*, 10:384–393, 1975.

58. P.D.V. v.d. Stok, F. v.d. Berk, R. Deckers, Y. v.d. Vijver, J.I.M. Botman, and C.J. Timmermans. Object-oriented design for accelerator control. In *RT93*, page Vancouver, May 1993.

59. J. P. C. Verhoosel. Off-line scheduling of hard real-time distributed systems using windows (extended abstract). In *Proceedings of the NATO ASI Workshop on Real-Time Computing*, St. Maarten, October 1992.

60. J. P. C. Verhoosel. Deterministic scheduling of distributed hard real-time systems using windows. In *Proceedings of the First IEEE Workshop on*

Parallel and Distributed Real-Time Systems. IEEE, Newport Beach, April 1993.

61. J. P. C. Verhoosel and et al. A static scheduling algorithm for distributed hard real-time systems. *Journal of Real-Time Systems*, pages 227–246, 1991.

62. J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, and A. D. Stoyenko. A model for assignment and pre-runtime scheduling of object-based, distributed real-time systems. *Journal of Real-Time Systems*, 8(1), January 1995.

63. J.P.C. Verhoosel, G. Yu, L.R. Welch, and D.K. Hammer. Pre-run-time scheduling for object-based, concurrent, real-time applications. In *Proceedings of the 2nd IEEE Workshop on Real-Time Applications*, pages 8–11, July 1994.

64. B. W. Weide, W. F. Ogden, and S. H. Zweben. Reusable software components. In M.C. Yovits, editor, *Advances in Computers*, volume 33, pages 1–65. Academic Press, 1991.

65. L. R. Welch. *Architectural Support for, and Parallel Execution of, Programs Constructed from Reusable Software Components.* PhD thesis, The Ohio State University, December 1990.

66. L. R. Welch. Architectural support for dynamic data distribution and dynamic scheduling. In *The Sixth Distributed Memory Computing Conference*, pages 83–89. IEEE, April 1991.

67. L. R. Welch. Assignment of ADT modules to processors. In *The International Parallel Processing Symposium.* IEEE, March 1992.

68. L. R. Welch. Cloning ADT modules to increase parallelism: Rationale and techniques. In *Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 430–437. IEEE, December 1993.

69. L. R. Welch. A parallel virtual machine for programs composed of abstract data types. *IEEE Transactions on Computers*, 43(11), November 1994.

70. L. R. Welch, A. D. Stoyenko, and S. Chen. Assignment of ADT modules with random neural networks. In *Proceedings of Hawaii International Conference on System Sciences*, pages 546–555. IEEE, January 1993.

71. L. R. Welch, G. Yu, B. Ravindran, F. Kurfess, J. Henriques, M. Wilson, M. W. Masters, and A. Samuel. Reverse engineering of computer-based navy systems. *International Journal of Software Engineering and Knowledge Engineering*, 1995 (in press).

72. L. R. Welch, G. Yu, J. Verhoosel, J. A. Haney, A. Samuel, and P. Ng. Metrics for evaluating concurrency in reengineered complex systems. *Annals of Software Engineering*, 1, 1995.

73. M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44. ACM SIGPLAN Notices 26(6), June 1991.

74. J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 19(2):139–154, February 1993.

75. J. Xu and D.L. Parnas. Pre-run-time scheduling of processes with exclusion relations on nested or overlapping critical sections. In *Proceedings of 11th Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC-92)*, pages 774–782. IEEE, April 1992.

76. G. Yu. Use of concurrency enhancement in off-line schedule construction. In *The Second Workshop on Parallel and Distributed Real-Time Systems*, pages 32–37. IEEE, Cancun, Mexico, April 28-29 1994.

77. G. Yu and L. R. Welch. A novel approach to off-line scheduling in real-time systems. *INFORMATICA Special Issue on Parallel and Distributed Real-Time Systems*, 19(1):71–83, February 1995.

78. G. Yu and L. R. Welch. Program dependence analysis for concurrency exploitation in programs composed of abstract data type modules. In *Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 66–73. IEEE, October Dallas, Texas, 1994.

79. G. Yu, L. R. Welch, W. Rossak, and A. D. Stoyenko. Automatic retrieval of formally specified real-time software components. In *Fifth Annual Workshop on Software Reuse*, October 1992.

80. W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, August 1987.

81. G. Zhu, L. Xie, and Z. Sun. A path-based method of parallelizing c++ programs. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation.* ACM SIGPLAN Notices 29(2), February 1994.