Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

ABSTRACT

SAFE CODE TRANSFORMATIONS FOR SPECULATIVE EXECUTION IN REAL-TIME SYSTEMS

by
Mohamed F. Younis

Although compiler optimization techniques are standard and successful in non-real-time systems, if naively applied, they can destroy safety guarantees and deadlines in hard real-time systems. For this reason, real-time systems developers have tended to avoid automatic compiler optimization of their code. However, real-time applications in several areas have been growing substantially in size and complexity in recent years. This size and complexity makes it impossible for real-time programmers to write optimal code, and consequently indicates a need for compiler optimization. Recently researchers have developed or modified analyses and transformations to improve performance without degrading worst-case execution times. Moreover, these optimization techniques can sometimes transform programs which may not meet constraints/deadlines, or which result in timeouts, into deadline-satisfying programs.

One such technique, speculative execution, also used for example in parallel computing and databases, can enhance performance by executing parts of the code whose execution may or may not be needed. In some cases, rollback is necessary if the computation turns out to be invalid. However, speculative execution must be applied carefully to real-time systems so that the worst-case execution path is not extended. Deterministic worst-case execution for satisfying hard real-time constraints, and speculative execution with rollback for improving average-case throughput, appear to lie on opposite ends of a spectrum of performance requirements and strategies.

Deterministic worst-case execution for satisfying hard real-time constraints, and speculative execution with rollback for improving average-case throughput,

appear to lie on opposite ends of a spectrum of performance requirements and strategies. Nonetheless, this thesis shows that there are situations in which speculative execution can improve the performance of a hard real-time system, either by enhancing average performance while not affecting the worst-case, or by actually decreasing the worst-case execution time. The thesis proposes a set of compiler transformation rules to identify opportunities for speculative execution and to transform the code. Proofs for semantic correctness and timeliness preservation are provided to verify safety of applying transformation rules to real-time systems. Moreover, an extensive experiment using simulation of randomly generated real-time programs have been conducted to evaluate applicability and profitability of speculative execution. The simulation results indicate that speculative execution improves average execution time and program timeliness. Finally, a prototype implementation is described in which these transformations can be evaluated for realistic applications.

SAFE CODE TRANSFORMATIONS FOR SPECULATIVE EXECUTION IN REAL-TIME SYSTEMS

by
Mohamed F. Younis

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 1997

APPROVAL PAGE
(Page 1 of 2)

SAFE CODE TRANSFORMATIONS FOR SPECULATIVE EXECUTION IN REAL-TIME SYSTEMS

Mohamed F. Younis

Dr. Alexander D. Stoyenko, Dissertation Advisor — Date
Director, Real-Time Computing Laboratory at NJIT
Associate Professor of Computer and Information Science, NJIT
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Thomas J. Marlowe, Committee Member — Date
Professor of Mathematics and Computer Science, Seton Hall University
Visiting Professor, Real-Time Computing Laboratory
& CIS Department at NJIT

Dr. Phillip A. Laplante, Committee Member — Date
Dean of Engineering and Technology, Burlington County College
Visiting Associate Professor, Real-Time Computing Laboratory
& CIS Department at NJIT

Dr. Rajiv Gupta, External Committee Member — Date
Associate Professor of Computer Science, University of Pittsburgh

APPROVAL PAGE
(Page 2 of 2)

SAFE CODE TRANSFORMATIONS FOR SPECULATIVE EXECUTION IN REAL-TIME SYSTEMS

Mohamed F. Younis

Dr. Bernard Lang, External Committee Member — Date
Member of Technical Staff, INRIA, Rocquencourt, France

Dr. James A. McHugh, Committee Member — Date
Associate Chairperson and Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Peter A. Ng, Committee Member — Date
Chairperson and Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Ami Silberman, Committee Member — Date
Assistant Professor of Computer and Information Science,
Member of The Real-Time Computing Laboratory at NJIT

BIOGRAPHICAL SKETCH

Author: Mohamed F. Younis

Degree: Doctor of Philosophy

Date: January 1997

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1997
- Master of Science in Computer Science,
  Alexandria University, Alexandria, Egypt, 1991
- Bachelor of Science in Computer Science,
  Alexandria University, Alexandria, Egypt, 1987

Major: Computer Science

Presentations and Publications:

G. Tsai, M. Younis, B. Ghahyazi, T. Marlowe, A. Stoyenko, "Evaluation of the Applicability and Profitability of Speculative Execution in Real-Time Programs", in the *Proceedings of The* 13^{th} *IFAC World Congress*, San Francisco, California, July 1996.

M. Younis, P. Sinha, T. Marlowe, A. Stoyenko, "Performance Enhancement of Various Real-Time Image Processing Techniques Via Speculative Execution", in the *Proceedings of the IS&T/SPIE Symposium on Electronic Imaging: Science and Technology*, San Jose, California, January 1996.

A. Stoyenko, T. Marlowe, M. Younis, "A Language for Complex Real-Time Systems", *The Computer Journal*, Vol. 38, No. 4, pp. 319-338, November 1995.

M. Younis, G. Tsai, T. Marlowe, A. Stoyenko, "Using Speculative Execution For Fault Tolerance in a Real-Time System", in the *Proceedings of the* 1^{st} *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, Ft. Lauderdale, Florida, November 1995.

BIOGRAPHICAL SKETCH
(Continued)

M. Younis, G. Tsai, T. Marlowe, A. Stoyenko, "Formal Verification of Transformation Rules for Speculative Execution in Real-Time Systems", in the *Proceedings of the* 20^{th} *IFIP/IFAC Workshop on Real-Time Programming (WRTP'95)*, Ft. Lauderdale, Florida, November 1995.

M. Younis, T. Marlowe, A. Stoyenko, "Compiler Transformations for Speculative Execution in a Real-Time System", in the *Proceedings of the* 15^{th} *IEEE Real-Time Systems Symposium (RTSS'94)*, San Juan, Puerto Rico, December 1994.

A. Stoyenko, T. Marlowe, M. Younis, A. Ganesh, C. Amaro, P. Laplante, A. Silberman, P. Sinha, "Towards A Language Paradigm for Construction and Development of Complex Computer Systems", in the *Proceedings of the IEEE Workshop on Composability of Fault-Resilient Real Time Systems*, San Juan, Puerto Rico, December 1994

M. Younis, T. Marlowe, A. Stoyenko, "Speculative Execution in a Real-Time System", in the *Proceedings of the* 2^{nd} *IEEE workshop on Real Time Applications (RTAW 94)*, Washington DC., July 1994.

M. Younis, A. Stoyenko, "Static Analysis Techniques and Speculative Execution in Real-Time Systems", an extended abstract in the *Proceedings of the ACM SIGPLAN workshop on Languages, Compilers, and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

H. Meske, M. Younis and W. Halang, "A Reduced Instruction Set for a Hard Real Time Architecture", in the *Proceedings of the ACM SIGARCH workshop on Architectures for Real-Time Applications (ISCA'94)*, Chicago IL., April 1994.

A. Stoyenko, L. Welch, P. Laplante, T. Marlowe, C. Amaro, B. Cheng, A. Ganesh, M. Harelick, X. Jin, M. Younis, and G. Yu, "A Platform For Complex real-Time Applications", in the *Proceedings of Complex Systems Engineering Synthesis and Assessment Workshop*, July 1993.

A. Stoyenko, T. Marlowe, W. Halang and M. Younis, "Enabling Efficient Schedulability Analysis Through Conditional Linking And Program Transformations", *Control Engineering Practice*, Vol. 1, No 1, pp. 85-105, 1993

M. Younis, M. Elattar, Y. Boutros, "Parallelization of the Finite Element Method to Solve Partial Differential Equations", *Alexandria University Academic Journal*, Vol. 5, No 1, pp. 112-122, 1992

M. Younis, *Parallel Solution of Partial Differential Equations on Transputers*, Masterś Thesis, Alexandria University, December 1991.

BIOGRAPHICAL SKETCH
(Continued)

M. Younis, G. Tsai, T. Marlowe, A. Stoyenko, "Statically Safe Speculative Execution for Real-Time Systems", submitted to *IEEE Transactions on Software Engineering.*

M. Younis, T. Marlowe, G. Tsai, A. Stoyenko, "Applying Compiler Optimization in Distributed Real-Time Systems", in the *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96)* (to appear).

A. Stoyenko, T. Marlowe, M. Younis, P. Petrov, "A Language Support Environment for Complex Distributed Real-Time Applications", submitted to *IEEE Parallel and Distributed Technology.*

A. Ganesh, T. Marlowe, A. Stoyenko, M. Younis, J. Salinas "Architecture and Language Support for Fault-tolerance in Complex Real-Time Systems", in the *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96)* (to appear).

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisors Prof. Alexander Stoyenko and Prof. Thomas Marlowe for their leadership and valuable advice. They taught me how to do research, guided me in my work and provided me with technical and moral support. I am very grateful to my external readers and the balance of my thesis committee for their time and the many excellent suggestions which led to a considerable improvement of this thesis. Thanks is also due to all members of the Real-Time Computing Laboratory at NJIT for their friendship and support as colleagues, as well as their constructive criticism and technical expertise. I am especially grateful to Dr. Grace Tsai, Behrooz Ghahyazi and Purnendu Sinha, who have been my coauthors in multiple publications.

I am truly indebted to the Office of Naval Research and, recently, also to the National Science Foundation for sponsoring our Lab's efforts for building a platform for designing and developing complex real-time systems. Working on this project has been a valuable experience and a strong assistance in the validation of my research. I would like also to thank these agencies for financially supporting my study and my trips to various conferences and research meetings. I would like to thank Plamen Petrov, Mehmet Akşu, Yuan Cheng, Chris Kline, Bin Liang, Robert Kates, Sun Kim, Bankim Patel, Yakov Khanin, Sergei Gorinsky and Jeff Venetta for their significant contributions to the project. In addition, I am very grateful to the Graduate Student Association at NJIT for their financial support of my presentations at multiple workshops and conferences.

I would like to thank Siemens Corporate Research, and in particular Drs. Paul Drongowski and Bob Schwanke, for two very productive summer internships that sharpened my research ability. In addition, I would like to thank the AlliedSignal Aerospace Microelectronics and Technology Center research team, Drs. David Shupe,

Jeff Zhou and Neeraj Suri for giving me the opportunity to join them and increasing my enthusiasm to wrap up my thesis.

My deep thanks to my wife Ghada for her support and assistance throughout my study. Finally, I would like to thank my parents who encouraged me all the way and blessed me with their prayers.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

"How is running a dedicated chemical process control program different from running a compiler on a time-sharing system? Apart from the obvious difference in their function, the two programs also differ in another fundamental way.

The time it takes the compiler to execute depends on such factors as the system load and program mix. Sometimes, the user must wait for a relatively long time for the compilation to complete, and at other times the compilation runs quickly.

Long compilations can be tolerated because there are no timing constraints associated with them. On the other hand, once the chemical process control program is started, it must not take arbitrarily long to execute. Such systems are called *real-time systems.*"[1]

Today there is a large and rapidly growing number of real-time applications. Such applications are drawn from different areas and use computer systems as control elements. These applications motivate research in the area of real-time systems. A special set of architectures, languages, compilers, and tools has been proposed in the literature to address the distinct requirements of real-time systems. This thesis mainly addresses compilation issues in real-time systems, and in particular on compiler-directed program transformations.

Compiler optimization techniques facilitate development and performance tuning of non-real-time systems[2]. Unfortunately, traditional compiler optimization can complicate the analysis and destroy the timing properties of real-time systems.

[1] from [38].

[2] While compiler optimization primarily speeds up execution of programs, it is can be used to enhance other features such as memory usage and code size.

That is why real-time systems developers tend to avoid automatic optimization of their code. However, real-time applications have grown in size and complexity which necessitates development of a set of compiler optimization transformations that tune performance without degrading worst-case execution times. Moreover, real-time systems have become distributed. Thus, the compiler transformations must consider more complicated issues, such as processor synchronization and the sharing of resources. This need motivates the current study of how to perform safe optimization and transformation of real-time programs.

Compiler optimization techniques include not only those transformations that enhance the average execution time of a single program, but also detection of opportunities for parallelism within the code in a parallel processing environment such as a distributed system. Detection of parallelism tends to be difficult, especially in real-time systems, due to timing constraints. A tool that can enhance the performance and expose additional parallelism without violating timing constraints can be of great benefit to a real-time application developer. For example, safe and beneficial opportunities for pure parallelism and speculative execution can be detected at compile-time. Speculative execution is an optimistic execution of parts of the code based on some assumptions about either the control flow or the values of variables. The assumptions are later validated and rollback may be required to preserve program semantics. Speculative execution can both enhance performance and increase parallelism.

This thesis mainly addresses safe use of machine-independent compiler optimization, parallelization and speculative execution techniques in real-time systems, with emphasis on the possibility of applying speculative execution to real-time programs without risking timeliness. A demonstration is provided for the use of speculative execution in limited but useful ways to safely improve the average-case, and sometimes even the worst-case, performance of a distributed

hard-real-time system. Compile-time analysis is used to verify both safety and profitability of speculative execution in real-time systems, relying on intensive static timing analysis to investigate the effect of rollback on worst-case execution time. The code is transformed to fork new processes to execute parts of the code speculatively on a shadow replica or on the same processor during a remote call or interleaving with the current process. This approach, to my best knowledge, has not been used before in real-time systems.

This chapter provides a motivation for our study of safe application of compiler optimization techniques to real-time systems, highlights the difficulties associated with performing such techniques and points out the contribution and the organization of the thesis. In the next section, those requirements that distinguish real-time systems from other, non-time critical systems, are presented. Then, compiler optimization without timing constraints is discussed in *Section* 1.2, and how timing constraints can complicate compiler optimization is illustrated in *Section* 1.3. *Section* 1.4 demonstrates how speculative execution can be useful in real-time systems and why it should be carefully applied. Finally, a summary of the major contribution of this work is provided, concluding with an outline of the balance of the dissertation.

1.1 Real-Time Systems Requirements

Real-time systems are distinguished from other types of computer systems by the explicit involvement of *time* expressed by two fundamental requirements: *timeliness* and *simultaneity* [38]. The semantics of a real-time system, and thus its correctness, involves not only the logical results of the computation, and the logical effect of communication, but also the meeting of deadlines on various aspects of the

system. The second requirement ensures simultaneous handling of external events in distributed and parallel processing of real-time programs.

Research in the area of real-time systems in the last few years confirms that real-time computing is not necessarily equivalent to high performance computing. Instead the requisite quality is not the temporal behavior itself, but rather the precise predictability and control of the timing behavior of even complex and distributed processes, leading to two other requirements of equal importance: *predictability* and *dependability*. Other requirements can be imposed by the nature of the application, such as security in military applications. The design of such systems has to provide enough reserve capacity and redundancy to be able to cope with extraordinary situations.

Soft and hard real-time systems are distinguished by the effects of a missed deadline. In soft real-time environments, costs rise with increasing lateness of results. On the other hand, no lateness can be tolerated in hard real-time environments, where late reactions may be either useless or dangerous. In other words, the costs of missing deadlines in hard real-time systems are infinitely high. Many applications have a mix of hard and soft constraints. Hard real-time constraints typically result from the physical laws governing the application.

For these reasons, common approaches to questions of performance evaluation in parallel processing systems, are inappropriate in the hard real-time domain. Thinking in probabilistic or statistical terms, the notion of fairness for the handling of competing requests, or the minimization of average reaction times cannot be used as an optimality criterion of system design. Instead, worst-case behavior, deadline satisfaction, maximum run-times, and maximum delays need to be considered.

For embedded real-time systems, moreover, optimality of processor utilization is a minor issue. Instead, costs have to be seen in the larger context of safety

requirements. For example, distributed real-time systems will usually have more than one real-time process running at a time. The execution of various processes should be synchronized to preserve the semantics of the system. This model of execution imposes more requirements to assure predictability and timeliness. Access to shared resources can be a source of unpredictability. Communication delays due to message routing and contention may affect a process's execution progress. Thus, the overall set of running processes should be pre-analyzed. An accurate timing tool should be used to consider all sources of overhead and contention. A feasible assignment of processes to processors, as well as a run-time scheduling policy, should exist under which every process can meet its deadline without violating the precedence of execution. Schedulability analysis [38, 89], determination at compile time of whether every process satisfies its timing constraints, becomes an important requirement for this type of real-time system. However, precise schedulability analysis is known to be NP-complete. In response, a set of techniques has been developed (see for example [97, 98]), which can reduce the scheduling complexity for many cases.

1.2 Compiler Optimization in the Absence of Timing Constraints

In this thesis, the issues of applying compiler optimization techniques to real-time systems without affecting timeliness are addressed. In this section, forms of compiler optimization, and various program representations and analyses commonly used to perform the optimization are discussed. The next section is devoted to illustrate problems of applying compiler optimization to real-time software and complexities introduced by timing constraints.

Compilation of programs for non-real-time applications typically involves local and global optimization of code to improve its expected running time. Data flow analysis techniques have proven to be important and beneficial for optimizing and

parallelizing programs in sequential, distributed, and parallel environments: for procedural languages such as Fortran or Pascal [2, 110]; for alternate language paradigms such as functional languages [34, 81]; and for explicitly parallel languages such as parallel Fortran and higher-level languages such as Hermes [102]. Recent results (for example [88]) have suggested that compiler optimization/parallelization can achieve results at least as good as hand-tailored code, and in some cases, much better.

Compile-time optimization, whether machine-independent or machine-dependent, should not affect the program semantics. This property is a safety requirement. In addition, optimization should be reasonably precise; that is, it should discover statically most opportunities which hold at run-time and provide gains with acceptable cost for the analysis. Both safety and precision require the use of only those paths which can be determined using reasonable analysis will be taken at run-time. These issues are discussed further in the *Chapter* 4.

In the absence of timing constraints, the most common objective function to be minimized with compiler optimization is the average-case execution time of a program. Code optimization techniques are generally applied after syntax analysis, usually both before and during code generation [2]. The techniques often consist of detecting patterns in the program and replacing these patterns by equivalent but more efficient constructs (all of our transformations fall into this category). These patterns may be local or global, and the replacement strategy may be machine-dependent or machine-independent. First, control flow analysis is used to extract the structure of the program. Then, improvable patterns are identified.

Code optimization can be divided into three interrelated areas. *Local optimization* is performed within a basic block of code. A basic block is a sequence of consecutive statements which may be entered only at the beginning and when entered is executed

in sequence without halt or possibility of branch except at the end [2]. *Loop optimization* is a transformation of code in a loop, e.g., lifting invariant statements or strength reduction of calculations. *Global optimization* is supported by data flow analysis – the determination at compile-time of information giving facts about communication and use of data. Data flow analysis can be seen as the transmission of useful relationships from all parts of the program to the places where the information can be of use. Data flow analysis includes intraprocedural analysis – analysis of a single function or procedure – and interprocedural (interprocess) analysis [60].

A more detailed discussion of the three forms of code optimization, as well as other forms of analysis used to enable optimization, is provided in *Appendix* A.

In this section, compiler optimization in the absence of timing constraints has been discussed. Timing constraints can introduce complex problems to applying compiler optimization to real-time software. The next section provides a discussion of the difficulties associated with performing code optimization of real-time programs in a single and a multi-process environment.

1.3 Real-Time Compiler Optimization

Timing constraints may make compiler optimization, discussed in the previous section, more complicated. Consider, for example, the code in Figure 1.1 which consists of a loop followed by a call to a critical region `crit(c)`.

Moving the invariant code `x := 5;` out of the loop will make the loop faster. Thus the call to the critical section (accessing shared resources) will be executed earlier. This may disturb the order in which requests are made for a shared resource, causing unpredictable delay time and may cause, as a result, another process to miss its deadline. Assume that before optimization this process will be the third in the queue following say, process B and C. After optimization, the call is issued more

ORIGINAL

```
while (i <= 100) do
  x := 5;
  j := f(i+x);
  i := i+1;
endwhile
call crit(C);
```

OPTIMIZED

```
x := 5;
while (i <= 100) do
  j := f(i+x);
  i := i+1;
endwhile
call crit(C);
```

Figure 1.1 A real-time optimization

quickly, so this process comes ahead of process C in the resource queue; which may increase the delay time for process C, causing it to miss its deadline.

Optimization has not often been applied to hard real-time programs, either to individual processes, or across processes. Real-time programmers have suspected that naive automatic optimization or parallelization prior to register allocation and resource scheduling can destroy safety guarantees and deadlines. While this suspicion is in fact correct, as shown in the above example, lack of optimization can also lead to missed deadlines. Consider again the code in *Figure* 1.1. The process may miss its deadline before optimization if it cannot execute fast enough to satisfy its deadline. By optimization, the loop invariant will be moved outside the loop boundaries and the size of the repeated block will be smaller. Thus, the loop runs faster and the process may be able to satisfy the deadline. However, optimization after register allocation and resource scheduling can destroy both allocation and schedule, while optimization before register allocation and scheduling may in fact worsen performance. In *Figure* 1.1, if optimization is performed after resource scheduling, the queue order of the shared resource may be changed (as the call in that process will be reached faster), and the previous schedule may no longer be valid.

In recent years, real-time applications have been growing substantially in size and complexity which makes it impossible for programmers to write optimal code, and consequently indicates a need for compiler optimization. Requiring optimization to be performed together with scheduling, instruction selection, register allocation, tends to make optimization a very hard problem. Marlowe and Masticola [59] have shown that even optimization for a system consisting of a single process may disturb timing constraints and may cause a deadline to be missed. In addition, optimization for explicitly parallel programs tends to be quite hard even without timing constraints [64].

As was shown earlier, proper optimization can sometimes transform programs which cannot meet constraints/deadlines or which result in timeouts into deadline-satisfying programs. Moreover, safe opportunities for parallelism can be detected that can, if carefully applied, enhance resource utilization and speed up execution. In addition, optimization of hard real-time programs has benefits even for real-time programs which are already running, and which can be proven to meet their timing constraints. For these programs, it is often preferable to reduce resource usage (time, space, or processors), especially in multiuser or multiprogramming environments. Not only do resources then become available to other users, but this may also make the programs more robust in the face of unpredictable system overload, as suggested by the scheduling-theoretic results of [11].

The following section shows how safe speculative execution can enhance average performance and generate opportunities for parallelism in real-time systems.

1.4 Speculative Execution in Real-Time Systems

In the previous section, the complexity of performing compiler optimization in real-time systems is illustrated. While there is a need for safe compiler optimization

for real-time code, it is very hard to apply optimization techniques without jeopardizing timeliness, even for simple models. This section includes an elaboration of the benefits of one interesting optimization technique, speculative execution, which is considered in depth in this thesis and the difficulties associated with applying it to real-time systems.

In systems without hard deadlines, expected execution times can be further reduced and parallelism can be increased by speculative execution.

Typically, speculative (or optimistic) execution [48, 103] requires rollbacks or restarts when the computation in progress is found to be based on assumptions which are later invalidated; rollback reads a checkpoint, and then replays as much of subsequent execution as is still valid, and begins execution (for a given process) when some step depends on changed information.

Speculative execution may: (1) execute a statement with outdated values, and need to retract the computation and re-execute it with the correct values, or (2) execute one branch of a conditional, and then need to retract that computation and execute a different branch, or none at all. Within this speculative execution, it may be possible to (3) make unnecessary calls or calls with invalid parameters, which will need to be retracted, if they have begun execution, or killed, if they have not.

Simple examples exist to show that, even when speculative execution provably improves expected performance, it can result in missed deadlines. In *Figure* 1.4, assume *exp* involves a call and takes time 8, *code block1* needs 10 units, *code block2* takes time 9, and the fork and copy each take time 2. If there is a 90% probability for *exp* to be true, the expected execution time for the original code (on the left) is 17.9 units, and becomes 12.7 for the transformed version (on the right). However, worst-case time has been extended by transformation from 18 in the original code to 19.

```
   ORIGINAL                        TRANSFORMED

if (exp)                        fork code block1
  code block1                   if (exp)
else                              copy results for block1
  code block2                   else
                                  code block2

/* Before transformation,       /* After transformation,
   the deadline is met */          deadline is possibly
                                   missed if exp is false */
```

Figure 1.2 Speculative execution can result in missed deadlines.

While this dissertation addresses safe use of machine-independent compiler optimization, parallelization and speculative execution techniques in real-time programs, speculative execution will be considered in depth. The thesis demonstrates how speculative execution can be used in limited but useful ways to safely improve the average-case, and sometimes even the worst-case, performance of a distributed hard-real-time system. The approach is not based on a specific architecture, but uses a number of architecture/operating system cost parameters. Compile-time analysis is used to detect both safety and profitability of speculative execution in real-time systems relying on intensive static timing analysis to investigate the effect of rollback on worst-case execution time. The code is transformed to fork new processes to execute parts of the code speculatively on a shadow replica or on the same processor during a remote call or interleaving with the current process. This approach, to the author's best knowledge, has not been used before in real-time systems.

Program transformations can be used to improve the timeliness, performance, and analyzability of real-time programs. However, to employ such transformations, they should be proven to be correct (both semantically and temporally), profitable, and automatable. To facilitate the use of speculative execution to real-time appli-

cations, which have grown in size and complexity, a set of compiler transformation rules is developed. The rules preserve not only program semantics but also timeliness [116], and can be incorporated into a real-time language compiler to be systematically applied. While applying these rules increase compilation overhead for real-time programs, this thesis shows that speculative execution pays off.

While the approach to speculative execution presented in thesis, and the related approach of [59], can be viewed as supporting primarily absolute performance improvement for real-time systems, information on deadlines and laxity can be used, both to enable additional transformations in the presence of slack, and to focus the efforts of the transformation system. In fact, even systems that are provably schedulable can benefit from such transformations. If the schedulability criterion is violated, and there are spare processors, speculative execution can be viewed as forking-off an additional process, presumably lowering the load per processor, and enabling the system to be scheduled. In addition, speculative execution can improve other properties of real-time systems, such as fault tolerance [117].

Speculative execution can be successful in computation-intensive complex systems, such as real-time imaging and multimedia. Although such applications have potential for parallelism, there are also opportunities for speculative execution [115]. Image filtration, for example, usually involves a lot of computation, while testing the quality of an image is time-consuming as well [19]. An image can be filtered speculatively on a shadow while quality tests are running. The same argument holds for edge detection. Moreover, morphological image processing [32] has a lot of potential for speculative execution. Construction of a structural element can be done speculatively while another element is being tried. Another application is image retrieval according to certain input or the occurrence of an event. The most complicated image can be retrieved and filtered speculatively on a shadow to shorten the worst-case execution.

1.5 Contribution

In this dissertation, we mainly study how to apply compiler optimization, in general, and speculative execution, in particular, to real-time systems. We identify safe and profitable opportunities for speculative execution at compile-time and transform the code accordingly. We have developed a set of transformation rules that can be plugged in compilers of most real-time languages. The speculative execution transformations have been integrated within a platform for developing complex real-time systems, being built at the Real-Time Computing Laboratory at NJIT and sponsored by the Office of Naval Research and the National Science Foundation. The platform is based on a new real-time language [99] and its tool support including an analysis and transformation engine. The speculative execution transformations has been implemented as a part of that engine. Detailed description of the platform is provided in *Chapter* 7. The contribution of this dissertation can be summarized as follows:

- We have developed techniques to detect safe and profitable speculative execution opportunities. We have defined a set of conditions that assure timeliness of real-time programs before enabling the transformation. We use compile-time analysis to justify safety and profitability of speculative execution. Safety is verified by investigating the effect of rollback on the worst-case execution time. The transformation is profitable when it speeds up the execution of the longest path of the program (refer to *Chapter* 4 for details).

- We have specified transformation rules that can be plugged in compilers of most real-time languages. The rules provide a set of preconditions, action and postconditions. Preconditions need to be verified to assure the preservation of program semantics and timeliness. The action part summarize changes in

the code, while postconditions reflect side effects resulting from the transformation. This format is proven to be very convenient for formal verification and implementation (refer to *Chapter* 4 for details).

- We have formally verified the safety of the transformation rules. We have used temporal logic to prove that the semantic of programs are preserved and the timing behavior is not worsen when applying the transformation rules (refer to *Chapter* 5 for details).

To validate our work empirically we have done the following:

- We have conducted an experiment to capture the effect of various properties of real-time programs that affect applicability and profitability of speculative execution. The experiment uses randomly generated real-time programs. We have examined the impact of the frequency of programming constructs, the size of blocks, and locality of variable references on the number of potential opportunities and performance gains due to speculative execution (refer to *Chapter* 6 for details).

- We have examined the usefulness of speculative execution in realistic applications. We have plugged in our transformation rules in a platform for developing complex real-time systems at the real-time computing laboratory at NJIT. The speculative execution transformations have been applied to a small number of simulated real-time applications, and shown to be beneficial for performance (refer to *Chapter* 7 for details).

1.6 Organization

This dissertation is organized as follows. In the next chapter, a real-time model which serves as a basis for this work is defined. *Chapter* 3 summarizes related

work. In *Chapter* 4, opportunities for speculative execution are identified, various safety issues affecting the applicability of speculative execution to real-time programs are elaborated, and a specification of compiler transformation rules for speculative execution is provided. These rules are formally verified for semantic correctness and preserving timeliness in *Chapter* 5. An experiment based on simulation have been conducted to capture various code properties that affect the number of feasible opportunities and performance gains of speculative execution. In *Chapter* 6, the design and results of this experiment are illustrated. A prototype implementation for the speculative execution compiler rules is described in *Chapter* 7, highlighting the applicability and usefulness of speculative execution in realistic application. Finally, *Chapter* 8 concludes this thesis and summarizes future research directions.

CHAPTER 2

THE REAL TIME MODEL

In the previous chapter, we motivated our study and defined the problem that this thesis is trying to address. In this chapter, a real-time model is defined for this work. In addition, definitions are provided for some of the terms used throughout the thesis. In the next section, assumptions about the hardware platform are stated, followed by a discussion of the assumed software environment. Schedulability analysis is illustrated in *Section* 2.3, followed by a discussion of high-level real-time programming language support. The discussion of the language model elaborates features that a language should provide to enable static analysis in the presence of timing constraints, as illustrated in the next chapter.

2.1 Hardware Environment

In this section, the thesis assumptions about the real-time hardware environment are stated.

Real-time hardware (for example [38, 63]) need not necessarily be very fast, but must provide predictable functionality enabling analysis of the system and fault-tolerance [25]. Issues like caching, direct memory access, virtual addressing, pipelining, or asynchronous communication protocols can cause nondeterminism, and consequently should be handled with care. In this thesis, it is assumed that the execution time of each machine instruction is known at compile-time. Moreover, it is assumed that the hardware does not introduce any unpredictably long delays into program execution. In the following section, the software component is defined.

2.2 Software Environment

Processes in real-time systems can be either periodic or aperiodic. Each process has a *frame* - the minimum period which corresponds to the maximum frequency of activation of that process. The frame is usually dictated by the external environment. The process can be activated periodically, by a signal from another process or an external activity, or at a specific time known at compile-time. Once activated, a process must complete its task before the end of the current frame (its deadline) and cannot be reactivated before the end of the frame (otherwise, the frame is not the minimum period). Processes can synchronize their execution. The kernel is responsible for serializing access to shared resources. A kernel call blocks a process until a desired shared resource is free, then it claims that resource and returns. All subsequent attempts to claim the same resource will block until the process with the resource executes another kernel call to release the resource. Synchronization primitives (for example, semaphores) can be used to implement this mutual exclusion. In this thesis, it is assumed that the kernel uses a suitable discipline to schedule processes, for example the disciplines described in [55, 66].

Traditional real-time systems have often taken the form either of a cyclic executive or of a relatively small number of independent, coarse-grained processes executed on a small number of processors and making use of a small number of mostly homogeneous resources. Current and future systems are expected to run on modern computer architectures, often parallel and distributed, and to utilize many heterogeneous resources. Consequently, techniques must be developed to identify parallel objects of appropriate granularity within real-time systems and to map these objects and their resource requests to parallel processes and resources, to facilitate such high performance objectives as short response times and balance of workload. In this thesis, it assumed that there is a suitable assignment tool, such as the tool in [100], within the real-time software environment to allocate such processes to processors.

Schedulability analysis, as illustrated in the following section, can provide at least some kind of prediction of execution behavior of a set of processes. This kind of analysis can help the programmer, as will be shown, to solve some of the allocation and scheduling problems at compile-time. In addition, compiler assistance may be used to collect additional information about the nature of the processes as an aid to the allocation and scheduling of processes, as illustrated in the next chapter.

2.3 Schedulability Analysis

The software components of modern real-time systems, as discussed in the previous section, are typically programmed in a high-level language with some functions possibly written in assembly code. As the software is written, the programmer attempts to follow the timing specifications of the system to the best of his or her ability. The resulting code is subjected to analysis for adherence to its critical timing constraints under all possible execution orders compatible with the scheduling discipline in use. This form of analysis, introduced by Stoyenko [89, 90, 91, 95] is commonly referred to as *schedulability analysis*. Schedulability analysis is also used for non-complex, scheduling-theoretic systems amenable to provably optimal rate-monotonic scheduling [58] to refer to its verification process, which typically involves checking of a simple set of constraints [30].

The schedulability analyzer consists of two parts, a partially language-dependent front end and a language-independent back end. The front end is incorporated into the code parser, and extracts timing information and calling information, and builds program trees. It computes the amount of time individual statements, subprograms, and process bodies take to execute in the absence of calls and contention. The front end has as an input table mapping statements to execution times. The back end is a separate program which analyzes the information

summarized in the generated program trees by the front end and predicts guaranteed response times for the entire real-time application.

The statistics generated by the schedulability analyzer tell the programmer whether or not the timing constraints are guaranteed to be met. In addition, it may provide the programmer with hints on problems or bottlenecks if the system fails one or more deadlines.

The accuracy of schedulability analysis depends on an accurate summary of timing information. However, finding precise solutions considering contention and branching in general is a NP-complete problem, and the cost can add significantly to the cost of program compilation. The NP-completeness arises in particular from the combinatorial explosion of possible execution orders in cases of processes sharing resources. As a result, schedulability analysis can either be (1) exact and efficient of analysis single process or multiple processes of simple form, or with highly constrained interactions [65, 72, 78, 108], (2) highly imprecise though efficient analysis of multiple process programs [57], or (3) nearly exact though highly inefficient analysis of some multiple processes [89, 95]. To combat some sources of combinatorial explosion, there has been work to reduce the cost of precise schedulability analysis, as for example [71, 96, 97, 98]. These are illustrated in the next chapter.

The next section provides a discussion of how schedulability analysis, among other requirements, motivated a new programming language paradigm for real-time high-level programming languages.

2.4 Real-Time Programming Languages

In the past, programmers for real-time applications have used assembly language to develop their programs. While assembly language provides enough control for

them to hand-optimize small processes, as the applications get larger, it becomes harder and more time-consuming both to develop and to optimize assembly code. Moreover, unstructured control flow and the use of address operators make automatic analyses difficult or impossible. Partly for this reason, the demand for high-level language programming for real-time applications has grown. Early designers of real-time languages took the natural approach of augmenting existing languages with real-time features. Later, a set of real-time languages was proposed structured around real-time requirements, such as Real-Time Euclid [51] and (to a lesser extent) Ada 9X [94]. Next, the requirements that real-time languages should support are discussed.

2.4.1 Requirements of Real-Time Languages

The requirements for real-time languages can be classified as: support for multiprogramming and distributed processing, expressibility of timing constraints, support for standard high-level language constructs while enabling schedulability analysis by avoiding or resolving constructs with unbounded execution time, and ability to describe non-functional constraints such as security and fault-tolerance.

Real-time software almost always involves multiprogramming. A real-time language must therefore support the process concept by providing process definition. It should allow concurrency and provide primitives for interprocess precedence, communication, and synchronization.

The most obvious requirement that a real-time language should satisfy is expressibility of a sufficiently powerful set of timing constraints to capture those imposed by the nature of time-critical applications. At a minimum, there should be constructs to express timing constraints on a process.

A real-time language should make sufficient provisions for schedulability analysis. Every program should be analyzable at compile-time to determine deadline

satisfaction during execution. The language should have no constructs that could take arbitrarily long to execute. For example, a general while-loop can lead to unpredictable execution times. While-loops are either removed from the language or require compile-time analysis or user assertions to provide an upper bound on iterations to bound execution time of the construct. Recursion can also be an obstacle for analyzing programs, and likewise may be disallowed or require compile-time knowledge of an upper bound on the depth of the recursion. Dynamic structures can have a similar effect, and are again disallowed or restricted, by a storage bound on their maximum size.

In addition to restrictions arising from timing constraints, there are generally other non-functional constraints. Real-time programs must in general be very reliable. Thus, a real-time language should be secure. Specifically, the language should have strong typing and structured constructs, and be modular as well as simple. There should be a high-level mechanism for exception handling to minimize the hardware-dependent part of the code that has to be implemented in assembly. This allows the portability of the programs to different platforms. Exceptions can also allow relaxation of constraints in abnormal situations, effectively supporting mode-change within the language.[1]

[1] A significant part of this discussion is derived from [92].

CHAPTER 3

RELATED WORK

A global requirement for all compiler transformations is to preserve the semantics of the programs. This property is termed the *safety* or *correctness* requirement. On the other hand, there should be some gain from applying them. It makes no sense to transform a program without enhancing some property of the analysis or the execution behavior. This requirement is termed *profitability.* For real-time programs, safety has a more restrictive definition: in addition, code transformations should not worsen the timing properties of the program. A program that meets all timing constraints should not be transformed to a one that fails its deadline. Thus evaluating the applicability of code transformations in real time systems requires an accurate estimation of execution time. Before performing the transformations, the effects on execution behavior must be studied. According to that investigation, the transformation may or may not be applied. The estimation of execution time can be based on a compile time prediction or monitored while testing the code. The better the accuracy of that estimate, the more confident we will be in transforming the code.

Usually compile-time analysis, including code transformations, is referred to as static or pre-run time analysis. Static analyses in real-time systems generally fall into four categories.

1. Code transformations guaranteed to preserve or enhance timing properties and to improve overall performance. These are generally safe forms of sequential and parallelizing compiler transformations, including in the latter category speculative execution. These transformations uniformly affect the executable code.

2. Partial evaluation and other forms of code specialization that largely support writing of high-level reusable code. Although transformations in this group result in changes in the executable code, their principal effect in working code is for the benefit of timing analysis. Transformations in this group provide support for predicting or monitoring execution behavior of systems.

3. Transformations to reduce the complexity of schedulability analysis. Recall from the discussion in the previous chapter that precise schedulability analysis is NP-complete. Transformations in this category attempt to decrease the complexity, as discussed in *Section* 3.5. These transformations may or may not change the code functionality.

4. Techniques to enhance the schedulability of the system, in the sense of trying to find a feasible schedule for a set of processes or extracting some useful properties about processes for the scheduler to use. These techniques seldom affect the code.

The work presented in this thesis falls primarily in the first category. It provides a study of how to apply various machine-independent compiler optimization techniques to real-time programs without jeopardizing timeliness. The thesis concentrates on safe and profitable use of speculative execution in real-time system. This chapter provides a summary of some of the previous work on static analysis of real-time systems and a comparison with the work presented in this thesis.

The chapter is organized by the goal of the analysis. However, some work can fit in more than one category. For example, in [36], the goal can be seen as enhancing schedulability and also as enhancing the average case performance by detecting more opportunities for interleaving execution. For another, the work in [35] can enhance utilization of resources, and also provides support for monitoring. The next two sections focus on the group 1 above, discussing previous work in compiler

optimization and speculative execution. The discussion of timing prediction and monitoring follows. Then, a discussion is provided of the previous work on enhancing schedulability. Finally, some work on efforts to enable efficient schedulability analysis is described.

3.1 Compiler Optimization

While much work has been done on compiler optimization, few papers consider real-time issues. Optimization can be categorized as either sequential program optimization or parallelization, and moreover, as machine-dependent or machine-independent. Here only machine-independent optimization is considered, assuming homogeneous memory. Using techniques related to the VPO approach [5, 27], on retargetable machine-dependent optimization, we may be able to extend our work detailed in the next chapter, especially in addressing issues of memory hierarchy. In this section, previous work on machine-independent optimization is considered, followed by efforts made to address real-time compiler optimization. Then, a discussion is provided about research on performing retargetable machine-dependent optimization.

Compiler optimization for sequential programs is discussed in [2], where most common machine-independent and machine-dependent optimization techniques are illustrated. An overview of parallelization techniques is presented in [110]. Both [2] and [110] address optimization in general without considering real-time systems.

Using a simple model for a class of hard real-time systems, Marlowe and Masticola [59] examine the applicability of classical source code transformations for both sequential optimization and parallel programming. They develop a notion of *safe* real-time code transformations and base their study on this safety property. A code transformation is a safe real-time transformation if it not only preserves

program semantics, but also preserves timing properties. They address only machine-independent optimization, using the intermediate code generated by the compiler. However, they address only deadline constraints, and do not consider the effect of the transformations in a multiprocess environment, nor do they have results on the applicability of such transformations.

Although we are principally interested in classic machine-independent optimizations, their safety for real-time systems appears to depend on memory hierarchy issues, and we may need to optimize at a level closer to target code. Work done on the VPO (Very Portable Optimizer) project at the University of Virginia and elsewhere addresses the use of retargetable machine-dependent optimization, which combines traditional machine-independent optimization with awareness of memory hierarchy issues and some machine-dependent optimizations. We would like to look in the future at applying this approach to real-time systems. In [5, 27], for example, an algorithm is presented to increase memory bandwidth for wide-bus machines by grouping fetch operations to get as many operands per memory read as possible. The advantage of machine-dependent global optimization is discussed in [13]. Two levels of intermediate code between the source code and the machine code are suggested: a high-level intermediate code used for machine-independent optimization, and an expanded low-level intermediate code. Most machine-dependent optimization can be performed on low-level intermediate code. In [14], low-level intermediate code for machine-dependent optimization is used to improve register allocation, in a portable manner.

In this thesis, machine-independent optimization is only considered, and leave the machine-dependent transformation as a future extension. In particular, the thesis focuses on the safe and profitable application of speculative execution to real-time software. In the next section, a discussion is provided for previous work on speculative execution which does not address real-time issues.

3.2 Speculative Execution

Speculative execution is an optimistic execution of parts of the code based on assumptions that need to be validated. Speculative execution has been substantially used for super-scalar and VLIW machines, for example [4, 8, 20, 21]. The model considered there is different from the one assumed in this thesis. The motivation is to come up with optimal instruction scheduling to achieve better performance and to decrease the overhead of rollback and recovery. Real-time issues are not addressed. Most approaches use machine-instruction-level speculative execution. The work presented in this thesis is not addressing that level of granularity, but is trying to extract opportunities at source code level. In the future, we may try to use a specific architecture and add instruction-level speculative execution. We are also looking at distributed real-time systems which may be running on a heterogeneous platform.

Speculative execution is also common in database management [105]. There has been work on speculative concurrency control and transaction management in real-time databases, such as [16, 17]. Redundancy is used to ensure that serializable executions are discovered and adopted as early as possible, to increase the likelihood of the timely commitment of transactions.

Moreover, speculation is used in early parallel implementations of logic and functional languages [39, 101]. With abundant processors, OR-parallelism is used by PROLOG interpreters to process in parallel the clauses for a predicate [83]. Anticipating a false value of the first clause, possibly unnecessary evaluation of the other clauses can be performed. However, this is a run-time mechanism, while our approach is to detect opportunities at compile-time and transform the code accordingly.

The use of rollback for synchronizing the execution of processes in distributed environments was introduced by David Jefferson in [48]. He defines the notion of

virtual time as a new paradigm for organizing and synchronizing distributed systems. Every process has its own local virtual clock. All messages output from one process are sent in virtual send time order but are not necessarily received in that order. However input messages to any process are read in virtual receive time order. He uses the Time Warp mechanism, a synchronization protocol distinguished by its reliance on lookahead-rollback, and implements rollback via antimessages. Every process continues execution, regardless of the virtual time of other communicating processes; if it encounters any message in its input queue with receive time-stamp less than its current virtual time, it performs a rollback to a suitable older state and sends antimessages to other processes to cancel messages previously sent during that period. He relies on a global control mechanism to detect global termination and to handle errors and I/O. When a process sends a command to an output device, output will only be physically performed if the global virtual time exceeds the virtual receive time of the message containing the command. After that point, no antimessages for the command can ever be generated and the output can be safely committed. Although he does not consider real-time processes, we may use his model as a base in considering speculative execution in multiprocess real-time environments.

The motivation in [103] is different. There, the problem of reconstructing a consistent state after a failure in a distributed environment is addressed. Optimistic recovery, an application-independent transparent recovery technique based on *dependency tracking*, is introduced. Dependency tracking entails each process to track its dependency on the states of the other processes with which it communicates. By recording such dependencies, it is possible to avoid unbounded cascades of rollbacks which may result in an attempt to find a consistent set of individual process checkpoints. To ensure that the externally visible behavior of the system is equivalent to some failure-free execution, all external messages are committed to the outside as soon as it is determined from dependency information that the states that

generated the messages will never need to be rolled back. Again, real-time issues are not addressed. We see dependency tracking as a possible technique to apply when we consider speculative execution in multiprocess real-time systems.

The possibility of optimistic execution of a process in the presence of more than one replica is studied in [33]. The purpose of process replication is to speed up the execution of a distributed application by reducing the communication delays with the replicated process. Optimistic algorithms are presented which guess whether the modification of a replica's state due to execution of a message can be performed long before applying the modification to the process' other replicas, without having the application observe the delayed consistency. If the guess is wrong, then execution of the message may have to be undone, but if the possibility that the guess is correct is sufficiently high, performance improves due to increased parallelism. The author considers both virtual time and dependency tracking as optimistic protocols. While his goal is close to that of this thesis, the approach is quite different. In this thesis, processes are not replicated; in addition, he does not consider real-time processes, and it is not clear that his technique applies without modification in this case.

A code replication technique to improve the accuracy of semi-static branch prediction is presented in [52]. The approach is to use profiling to collect information about the correlation between the subsequent outcomes of a single branch, especially for intra-loop branches. Considering that history (profiling data) at compile-time, it is possible to enable speculative execution based on that history. The disadvantage of this approach is the increase in code size which may have negative impact on instruction cache miss rate; thus there needs be a cost function which takes both execution speed and code size into account. In addition, it requires two-phase compilation – once for the profiling run, and once using that information for eventual execution.

Yamana et al. [112] showed the effectiveness of speculative execution of conditional branches in enhancing the average performance of Fortran programs running on multiprocessor platforms. They use static single assignment [2] to avoid race conditions in their shared memory model. They duplicate code on conditional branches to increase the effectiveness of speculative execution. They proposed a distributed control mechanism with: a global data matcher to trigger the execution of the speculative macrotasks (threads) upon the fulfillment of all data preconditions, a broadcasting system that report the progress in execution, and a dynamic task allocator to assign macrotasks to under-utilized processors. However, safety and profitability of speculative execution is not justified before transformations and during the execution. In real-time systems, it should be ensured that enhancing average performance does not jeopardize timeliness. Our approach is to statically verify the safety and profitability of speculative execution before transforming the code.

Rauchwerger and Padua [84] use speculative execution with run-time tests to enable parallelization of loops with statically unknown cross-iteration dependence. Their approach is to optimistically transform the loop so that all iterations are to be executed in parallel on different processors. A run-time data dependence test is to be applied to determine if there had been cross-iteration dependences during the execution. If the test fails, the loop is re-executed serially and the original execution time is extended by the time of the test. Thus, we see that this technique is inappropriate for real-time systems.

Automatic parallelization of *while* loops through transformation into equivalent *for* loops have been proposed by Wu and Lewis [111]. The idea is to extract a variable that can serve as a loop index. Although this technique can handle certain types of *while* loops, others cannot be transformed (*true while* loops). Collard [24] proposed a technique to automatically parallelize *true while* loops using speculative execution.

However, his approach is restricted to a single *while* loop surrounding *for* loops that perform vector-related operations. The technique creates multiple shadow copies of arrays and introduce another dimension indexed by a shadow index. Our approach does not have such restriction either on the loop control structure or the loop body. Moreover, the notion of safety in our transformations includes timeliness, which substantially increases the difficulty of the problem.

3.3 Enhancing the Prediction of Execution Time

As illustrated at the beginning of this chapter, code transformation for real-time systems must rely on an accurate timing analysis. In this section, some of the work done on static prediction of execution time is discussed, as well as attempts to use compiler support to enable monitoring execution of real-time processes. Some work has addressed the calculation of the execution time of a single process in isolation, while others have studied expected timing behavior in the presence of multiple processes competing for shared resources. Some approaches assume program annotations to obtain better estimates of the execution time, for example [37]. Others use perturbation analysis techniques to propose locations in the code at which to perform run-time monitoring activities, like [87]. On the other hand, some consider performing a simulation to reflect changes in timing behavior due to any code transformations [107], or due to sources of unpredictability such as cache memory and pipelining [68, 70].

In [71, 72], the aim is to relax the restrictions placed on the use of high-level constructs, such as recursion, loops, and dynamic data structures in real-time software, and to obtain better estimates for each execution instance, instead of worst-case estimates over all instances. The technique is based on partial evaluation. Using information available at compile-time about the execution environment and/or values

of variables (e.g., the length of input arrays), a residual program specialized for that environment can be derived by partial evaluation. Analyzing the residual program, a more realistic upper bound on execution time can be determined.

A different approach can be found in [78]. The first step is to perform basic prediction of the execution time statically, based on a simple timing schema for source-level language constructs. Then using user-provided information, dynamic path analysis allows refinement of the original predictions by eliminating paths and decomposing the possible execution behaviors in a path-wise manner. Aspects of this approach are closely related to [96].

The objective in [37], similar to the above, is to refine execution time estimates of real-time applications. Refined estimates can be used at run-time to achieve better resource utilization and early failure detection and recovery. The approach is to detect, at compile-time, correlation between execution of a statement or a block of code and the evaluation of a branch condition, between the execution of a statement outside a loop and the number of loop iterations, between the call site of a procedure and the evaluation of a branch in that procedure, or between the execution of a statement and creation of a task in a parallel program. Based upon the execution path followed and the correlation information, the worst execution time of the remainder of the task can be estimated. The scheduler can consult that estimate to perform such adaptations as may be required to ensure deadlines, to pre-schedule other related processes, and to pre-allocate resources.

The approach of [56] is to implicitly consider program paths without explicitly enumerating them. The problem of determining worst execution bounds is converted to solving an integer programming problem. Basic blocks are analyzed at compile-time and the execution bounds are computed based on a machine model. Using integer programming, an upper bound on the number of times a basic block is visited

can be computed. User annotations can be incorporated as a set of constraints. While each integer programming problem can in the worst-case take exponential time, the authors argue that in practice it takes time similar to that for solving a single linear programming program.

Perturbation analysis, a technique which identifies the situations in which run-time monitoring activities can be performed non-intrusively, has been proposed in [87]. The techniques identify the idle time available during the execution of a task and schedule monitoring tasks during these times. They partition the monitoring work among various points at which idle time is available. The approach is based on viewing a real-time application as a series of execution spans, delineated by input points, at which the computation must wait to receive data. Idle times occur during those periods when the current execution is complete and the computation is suspended at an input point. They view the resulting idle time as the amount of monitoring work that can be absorbed without affecting the program's ability to meet deadlines. They rely on static timing analysis during compilation and rely on user annotations for monitoring requests to analyze the possibility of monitoring without affecting worst case execution time.

The problem addressed in [107] is different. The author tries to calculate accurate source-level execution time bounds for real-time programs in the presence of code improving transformation. The compiler builds a timing tree reflecting the execution time of the basic constructs in the program. The tree is then modified when program optimization or code motion is performed, so that it is possible to see the effect on the execution time.

A method, called Static Cache Simulation (SCS), is introduced in [68] to enable using instruction caching without sacrificing predictability. The approach is to use control flow information provided by the back-end of a compiler to simulate cache

behavior at compile-time. Knowing that behavior it becomes possible to analyze worst-case execution timing of the program while gaining performance enhancement using instruction cashing.

This idea has been extended to support the use of software-based cache partitioning to maintain predictability of execution time within preemptive real-time systems [67]. Tasks will be associated with distinct cache partitions. Compiler transformations are introduced to provide instruction and data partitioning. Separate object files are generated for each code and data partition. The linker combines these objects files into an executable. Static cache simulation can still be performed for individual tasks. However, such transformations introduces new code that may increase the worst-case execution time. Evidence of the usefulness of such technique is pointed out as future work.

In the same spirit as SCS, a portable pipeline simulator compiler is proposed in [70]. There, they try to predict pipeline behavior for uniprocessor RISC architectures containing multiple functional units, multicycle operations and out-of-order instruction execution. They use an architecture description file to model the pipeline. The simulator compiler analyzes the real-time program in that architecture assembly language, and provides the worst-case execution of blocks of interest within the code.

An approach to integrate the timing analysis of pipelining and instruction caching is presented in [41]. Static cache simulation is used to categorize the caching behavior of each instruction of a given program. The caching behavior of instructions within a path is considered to predict the pipeline performance of that path. The performance of various paths are integrated to predict the worst-case execution time of the program.

Huang and Liu [46] address the unpredictability of programs execution due the use of direct memory access. They suggest an algorithm to give a bound on

the worst-case execution times of the concurrent execution of CPU instructions and cycle-stealing DMA I/O operations. This analysis is applicable in absence of cache memory and pipeline operations.

A heuristic algorithm to determine an upper bound on the response time of each process in a distributed real-time environment is presented in [57]. The objective is to determine the worst-case blockage due to competition for shared resources (processors, critical sections, devices, communication links). The approach starts with each process simultaneously blocking every process that could block. From this, they then remove impossible blockage combinations, corresponding for example to two processes executing the same critical section; thus the remaining blockages are always an upper bound on the worst-case blockage.

3.4 Compiler Transformations to Enhance Schedulability

There has been a great deal of work to enhance the schedulability of real-time processes using compiler transformations. Some generate useful decision support for the on-line scheduler, like [36]; others consider a priori fixed specific real-time scheduling algorithm, for example [45, 31].

In [36], a compile-time technique is presented to enable interleaving the execution of tasks on a single processor and/or overlapping the execution of tasks on multiple processors using a restricted resource contention model. They suggest a new task graph representation, called the compact task graph, used to aid in the scheduling of a set of communicating periodic tasks. Assuming availability of necessary resources, busy-idle execution profiles of the real-time tasks [87, 97], discussed earlier in this chapter, are computed during compilation. The intention is to expose the potential for parallelism across tasks, as well as idle times that may be encountered with a task, in the compact task graph. By providing this information

within the compact task graph without splitting tasks, the authors argue that it will be suitable for achieving efficient on-line scheduling.

A different approach to support on-line schedulers is introduced in [35]. First, compiler-based techniques classify the application code on the basis of predictability and monotonicity. Then, those techniques are applied to introduce measurement code fragments at selected points in the application code and to store monitoring data. The results of run-time measurements can be used to dynamically adapt worst-case schedules. The approach is based on the ability to reorder the code during compilation, so that parts of the code with unpredictable execution times are executed earlier. Using run-time measures of actual execution, the deviations from anticipated worst-case execution times can be considered by the scheduler for a remedy if a process will miss its deadline, or for accommodating additional tasks if there is slack time. The goal is to enhance utilization of resources and speed detection and recovery from failure.

An algorithm to achieve consistency between the program's worst-case execution time and its real-time requirements is presented in [31, 45]. They use a language model based on time-constrained relationships between observable events. Then, they apply compiler transformations to sequential programs to move unobservable code so that the task can comply with its timing requirements. First, the code is translated into single static assignment form (SSA) [110], followed by decomposition it into blocks. Then a variant of section-wise trace scheduling is applied to attain feasibility or to decide that the program is infeasible.

Based on the same language model as [31, 45], a task transformation technique for control domain applications driven by rate-monotonic scheduling [58] has been presented in [30, 44]. The objective of the transformations is to enhance the schedulability of the system by transforming unschedulable tasks into multiple threads.

Another approach to enhance schedulability in a preemptive real-time scheduling environment is discussed in [85]. They rely on compiler assistance to reduce the overhead due to context switching for preemptive scheduling in real-time systems. The method introduced in their paper tries to detect points in the program where only a small subset of the registers are live. By performing context switching at those points in the program, it will be possible to avoid saving and restoring irrelevant registers, and consequently to reduce the time of context switches. They push this idea further by introducing a machine-dependent optimization technique, called register remapping, to provide more fast context switch points. To avoid degrading the performance of the code, they rely on hardware support to identify those points for the scheduler.

3.5 Enabling Efficient Schedulability Analysis

Schedulability analysis, as discussed in the previous chapter, refers to static determination of the satisfaction of timing-constraints by real-time programs. Precise compile-time verification of execution constraints on timing is known to be NP-complete. Practical schedulability analysis of large real-time applications will thus require tools or analytical techniques to reduce the expected problem size and computation time. In this section, some of the work done to enable efficient schedulability analysis is discussed.

A polynomial-time code transformation to simplify schedulability analysis of parallel real-time programs has been presented in [97]. A restricted subset of Real-Time Euclid [51] is used as a language model. A restricted form of shared resource contention of processes to simplify the analysis is assumed, as also assumed later by the similar approach of [36]. All resource requests participating in a non-idling resource interval are released together, when the last request is finished. Moreover,

the resource scheduler enforces statically pre-computed non-idling resource interval sizes. Using that model, they introduce clustering transformations, via attribute grammars [2]. When critical sections occur on branches of a condition, the clustering algorithm inserts fixed delays into some branch to make accesses to a critical section happen at the same time as on other branches. Thus, the complexity is reduced by a factor of two (for each such transformation) while the process will have the same effect on other processes regardless of the branch it takes.

The work in [98] has a different approach to decreasing the number of paths to be considered by the schedulability analyzer. The approach is to detect non-executable paths by linking execution of conditional branches in various parts of the program. They combine that with the clustering transformation discussed above, and have some positive experimental results illustrating the applicability of these techniques in reducing problem complexity. While the technique of [35] is somewhat similar, that technique aims at supporting non-intrusive monitoring.

CHAPTER 4

DERIVING REAL-TIME COMPILER RULES

In previous chapters, the problem of applying compiler optimization techniques to real-time systems, and previous efforts of different researchers have been illustrated. In this chapter, we present our approach to solving that problem. This chapter shows that it is possible in many cases to apply compiler optimization and parallelization techniques without affecting the timing constraints of the system. Thus, real-time systems programmers can use compiler optimization to tune the performance of their systems and enhance resource utilization.

A set of transformation rules is introduced that can be applied to perform optimization, parallelization and speculative execution. The applicability and safety of the transformations are specified through a set of preconditions. Preconditions include structural, dependence and blocking conditions to preserve the program semantics, and timing conditions to avoid extending worst-case execution time. The transformation engine within the compiler makes sure that all these preconditions are verified before modifying the code. The engine interacts with the timing tool and uses data flow analysis techniques as well as the program control flow graph. Once the transformation is proven to be safe and profitable, the engine will make the appropriate code modifications.

In the next section, the adapted model for this approach is stated and the syntax of the rules is defined as well as some of the abbreviations used; a discussion of optimization and parallelization rules follows. The discussion of speculative execution begins with detection of opportunities for speculative execution. Then an example of code transformation is shown, illustrating some of the issues to be considered, as

well as problems to be overcome. Finally, compiler rules for speculative executions are provided.

4.1 The Model and Form of Rules

In this chapter rules for a number of compiler transformations for optimization, parallelization and speculative execution are discussed. This section starts with providing definitions and assumptions needed within the model, as well as representation of the rules.

4.1.1 A Problem Model

A real-time model similar to the one presented in Chapter 2 is adapted. A set of periodic top-level processes are assumed, each with a deadline, invoking methods of a set of objects governing resources and data. The application runs on an arbitrary network of processors. Objects and processes are assigned to processors at compile time.

The analysis relies on an expressive real-time language for all kinds of timing constraints. The language does not allow any unpredictable constructs: there are no dynamic structures, all loops have an upper bound on the number of iterations, and there is no recursion. Conceptually, a program in this language may have resulted from source-to-source translation of a program with more general loops and with limited recursion [99, 22]. However, it is assumed that the language allows concurrency and interprocess synchronization. The execution time of a machine instruction is known. Moreover, there should be an upper bound on communication delays.

Throughout the thesis we use the following data dependence terminology describing dependences between the code of S and other code in P:

- *True* or flow dependence: Value of a variable x, defined in S, reaches use in P.
- *Anti*-dependence: Value of a variable x used (read) in S is subsequently redefined by a definition in P.
- *Output* dependence: Value of a variable x set in S is overwritten by the definition in P.
- *Input* dependence: Value of a variable x used in S is subsequently be used next in P.
- *Control* dependence: The execution of P is controlled by the value of a predicate in S.
- *Resource* dependence: Resource R (console, monitor, file, ...) is accessed in S, and may be accessed next in P, and resource R is ordered (reordering accesses to R has significant and observable semantic effects).
- *Data* dependence: true, anti, and output dependence (input dependences typically matter only in the presence of memory hierarchies).

4.1.2 Representation of the Rules

An axiomatic specification approach is used that includes both preconditions and postconditions to denote the execution before and after applying a transformation for speculative execution. There are other approaches, for example [106, 109], for specifying data dependence and control flow conditions. However, in their current form they are not suitable for real-time systems, since compiler transformations of real-time programs cannot ignore timing constraints and resources access. The rules are standard Hoare triples [43]:

$$(precondition, action, postcondition)$$

In each rule, the code S in a procedure/method P is considered. The set of preconditions identifies applicability, correctness, and profitability, and is decomposed into the following subsets:

- One invariant condition: except as provided below, certain types of *blocking statements*, for which linearizability is important or retraction is impossible (e.g., I/O, creation/destruction of resources, exceptions with persistent effects, possible errors), do not occur in S. It is assumed that resource dependences have been captured in *Blocking* or *Ordered* constraints on resource access (see below).

- Structural conditions: Syntactic flow-graph conditions on S.

- Dependence conditions: Summarize the dependences between the code of S and other code segments in P.

- Blocking conditions: Additional blocking or unblocking information, possibly guarded by their own preconditions.

- Timing rules: Needed to determine the profitability of the transformation.

The following information is used in specifying conditions:

- The standard PDG decomposition of dependence into control dependence, true (flow) dependence, anti-dependence, output dependence, and input dependence, and for dependences inside loops, into loop-dependent and loop-independent dependences

- $Vars(S)$ = the set of variables referenced in S.

- $Mod(S)$ = the set of variables modified directly or indirectly in S.

- $Pres(S)$ = the set of variables whose definitions must be preserved through S.
- $Calls(S)$ = the set of method calls in S.
- $Blocking(L)$ is true if L is blocking (that is, concurrent accesses to L are forbidden); $Ordered(L)$ is true if the order of accesses to L is observable.
- For a method M, $TCalls(M)$ = the set of methods/procedures transitively calling M.

In addition, the following timing functions are assumed: first, given a set of variables $Vars$, the functions t_c and t_r are assumed to give the time to copy and restore that set of variables; second, $Time(S)$ returns an estimate for the worst-case execution time of S, which may be a code segment, a procedure, or a method. Also t_f and t_j are used for fork time and join time respectively (both include communication delays).

After stating our model and assumptions, we proceed with our contribution. In the next section, safe compiler optimization transformation rules are presented, followed by an illustration of those for parallelization and speculative execution.

4.2 Safe Compiler Optimization Rules

This section includes a presentation of the contribution of this thesis to the problem of performing profitable compiler optimization for real-time systems without jeopardizing the safety of the system.

As shown in the previous chapter, there has been a very little work that address real-time issues while applying compiler optimization. An attempt to classify classical machine-independent compiler optimization according to safety in a single process context is presented in [59]. A simplified model of real-time processes is

assumed allowing only the use of homogeneous memory and a subset of possible timing constraints. In our opinion, four research directions can be explored. The first is to generalize the set of timing constraints allowed. The second is to extend the model to support multi-process analysis. The third direction is to get closer to the machine-dependent optimization including the effect of memory hierarchy. The fourth and final direction is to construct and implement transformation rules to experiment the effects on performance, timeliness and predictability of real-time programs. In this dissertation, we are moving in a combination of the first and last directions. We are also exploring the second, but that work does not form part of this thesis. We are building a compiler optimization transformation engine based on the same simple model. We leave the other directions as a future extension.

```
   ORIGINAL                    OPTIMIZED

if (s1)                      s2;
  s2;                        if (s1)
  s3;                         s3;
else                        else
  s2;                          s4;
  s4;                       endif
endif                       s5;
s5;
```

Figure 4.1 Unsafe code hoisting

In this section, one example of the optimization transformation rules is illustrated, namely for code hoisting and sinking. A list of rules, can be found in *Appendix* B.

If a conditional statement has identical code in its then and `else` branches, it may be possible to optimize the program size by *hoisting* the identical code before the conditional. However, code hoisting may be unsafe if the code is hoisted past events. The same holds for code sinking. For example, consider the code segment

RULE:	Code Hoisting
Preconditions:	Structural: (1) $S = S_1$; if (exp) then $S_2 S_e S_2'$ else $S_3 S_e S_3'$. (2) Neither S_2 nor S_3 contains any critical sections or access to a shared resource. Dependence: (3) There is no dependence from S_2 or S_3 to S_e. (4) There is no data or resource dependence from exp to S_e. Timing: (5) Either branch will meet its deadline.
Action:	Transform S into S_e (if (exp) then $S_2 S_2'$ else $S_3 S_3'$)
Postcondition:	No deadline will be missed.
Comment:	Will not interfere with dependences from S_e to S_2' or S_3' since otherwise would have been output or resource dependences from S_2 or S_3 to S_e.

Figure 4.2 Rule: safe code hoisting

in *Figure* 4.1. If $s1$ contains a call to a critical section, then moving $s2$ before the if-statement will delay the call in $s1$ causing a miss of a deadline. The transformation rules for code hoisting and sinking are shown in *Figure* 4.2, and 4.3 respectively.

RULE:	Code Sinking
Preconditions:	Structural: (1) $S = S_1$; if (*exp*) then $S_2 S_e S_2'$ else $S_3 S_e S_3'$. (2) Neither S_2' nor S_3' contains any critical sections or access to a shared resource. Dependence: (3) Neither S_2' nor S_3' depends on S_e . Timing: (4) Either branch will meet its deadline.
Action:	Transform S into (if (*exp*) then $S_2 S_2'$ else $S_3 S_3'$) S_e
Postcondition:	No deadline will be missed.
Comment:	Will ordinarily prefer to use Code Hoisting when both are applicable.

Figure 4.3 Rule: safe code sinking

4.3 Safe Parallelization of Real-Time Programs

Wolfe [110] gives a series of source code transformations which may be useful in exposing parallelism and vector operations in a program. This thesis address only those parallelization transformations that detect segments in the code that can run simultaneously on different processors without violating the real-time safety requirement for transformations.

Consider the following two segments of code:

S_1 ; S_2 ;

Assume that there are no data dependencies from S_1 to S_2. By inspection, there are no control dependencies. Moreover, assume, without lost of generality, that $AvgTime(S_1) < AvgTime(S_2)$, and that the time cost of spawning and initializing S_2 (fork time t_f) and obtaining the results (join time t_j) is less than $AvgTime(S_2)$. That code can be transformed so that S_2 can run in parallel with S_1 on a different processor.

That transformation will be safe if S_2 does not have a call to a critical section. On the other hand, it may be also safe if making that call from S_2 earlier is not going to disturb the queue order of that critical section. Multi-process analysis or scheduler pragmas may be required to prove the safety of the transformation in that case. One compiler transformation rule for this parallelization transformation is shown in *Figure* 4.4.

In the following section, another approach to extracting parallelism in real-time programs is presented. The approach is based on investigating the opportunities for safe and profitable speculative execution in real-time systems.

4.4 Opportunities for Speculative Execution

Unconditional parallelism is infeasible, and speculative execution will be required, when there is a true dependence (or in certain cases of output or resource dependence) from the callee to subsequent code in the caller (or in subsequent called procedures). Recall that speculative execution occurs when it is required to execute code without being certain whether the code will execute, or will not be sure of the initial values of variables. It is useful to distinguish these two cases. We say that one block S_2 is value dependent on another S_1 if values computed in S_1 affect the initial values in S_2. The block S_2 is predicate dependent on S_1 if values computed in S_1 can affect whether S_2 executes, but not the initial values if it does.

RULE:	PARALLELIZATION
Preconditions:	Structural: (1) $S = (S_1; S_2)$ is a single-exit code region. (2) Neither S_1 nor S_2 contains any critical sections or access to a shared resource. Dependence: (3) $Vars(S_2) \cap Mod(S_1) = \phi$ (There is no data dependency in S_2 on S_1) (4) There is no control dependency between S_2 and S_1 Blocking: (5) For each method M in $TCalls(Calls(S_1)) \cap TCalls(Calls(S_2))$, $not(Ordered(M))$. (Incorrectly or prematurely executing any such statement has a permanent and invalidating effect on the environment.) Timing: (6) $AvgTime(S_2) > t_f + t_j$. (7) $MAX(Time'(S_1), Time'(S_2)) + t_f + t_j < Time'(S_1) + Time'(S_2)$. (Useful work can be done; worst-case time does not increase.)
Action:	Execute S_2 in parallel with S_1. Insert synchronization between $exit(S_1)$ and $exit(S_2)$.
Postcondition:	S has completed without missing its deadline. State is as if execution had been sequential. The average execution time is reduced.

Figure 4.4 A parallelization transformation rule

It can be seen that S_2 is value dependent on S_1 only if there is some transitive true dependence, or an input or output dependence interacting with the environment, from S_1 to S_2. On the other hand, S_2 is predicate dependent on S_1 if any transitive dependence from S_1 to S_2 involves a control dependence. In this section, the detection of opportunities for safe and profitable speculative execution is discussed.

4.4.1 Opportunities of Speculatively Executing Conditionals

Assume that there is a call at a branch point (for simplicity exactly two branches are assumed) and the code is of the form $p : if\ (C)\ then\ S_2\ else\ S_3$, where C is a call being executed on another processor. If the current state at p is stored (and possibly later retrieve), there will be a cost of c_s for the store, and c_r for the restore.

If the execution time of S_2 (T_{S_2}) dominates that of S_3 (T_{S_3}), and $T_{S_2} - T_{S_3} > c_r$, and further, some initial segment of S_2 is not data dependent on an *out* parameter of C, then S_2 can be executed speculatively, abandoning the computation and restoring prior state only if the returned value indicates that S_3 should have been chosen. This will almost invariably be the case in dealing with an `if-then` statement, since S_3 is the empty statement. However, for the transformation to be useful, it requires that the evaluation of C (possibly together with some prior statements on which s is not data dependent) be time-consuming.

If one branch has little or no effect on state, so that the restore is inexpensive, and that branch has some initial segment not data dependent on C, that branch can be speculatively executed. (If both branches have this property, the one with the longer execution time will be used.) Furthermore, if there is a data dependence on a value modified in C, that branch can be speculatively executed and the execution can be stopped at the point when that value needs to be used, provided that this is profitable.

4.4.2 Opportunities of Speculatively Executing While Loops

While the model of [89] allows only constant-count loops with compile-time bounds, this can with care be extended to allow `while` loops with a compile-time-provable bound on iterations, or equivalently, constant-count loops with exits. In parallelizing compilers, detecting parallelizable loop iterations, and distributing iterations among processors, is a major source of improved performance [50, 84, 110]. For `while` loops, this may involve speculative execution of some number of iterations, saving the state after each. In speculative execution, the next loop body may be evaluated in parallel with a call late in the previous body, where the loop condition depends on the return value [49]. For example, in the code block p : *while* (C) *do* S_2 , where C is a call being executed on another processor, execution of the loop body S_2 (or part of it) can be started during the evaluation of the call C, undoing all updates to the variables if the evaluation of the condition results in termination.

One particular subcase which proves interesting is the case in which iterations modify distinct locations, as, for example, in array-oriented programs. In this case, the original values can be remembered, allowing iterations to proceed, and restoring precisely the values which have been written by speculatively-executed iterations which do not in fact occur. This technique could possibly be generalized using an approach such as last-write-trees [7].

Again, attention should be paid to the worst-case scenario. The cost of rollback must be estimated. The transformations may not be performed if they endanger satisfaction of timing constraints imposed on the program.

4.4.3 Opportunities for Shadow Execution

The technique of *shadow execution*, modifying a copy of the store during speculative execution and copying into actual storage upon commitment, is an alternative to

checkpoint-and-restore strategy frequently used in databases [17, 18, 105]. That is, checkpoint-and-restore copies state, then modifies the original state, and ends by either (upon commit) discarding the copy, or (upon abort) copying the checkpoint back to the store. In contrast, shadow execution copies state and executes, thus *modifying the copy*, and ends by (upon abort) discarding the copy or (upon commit) copying any changed values to the original.

The same technique presents additional opportunities for speculative execution in real-time systems. Typically, discarding modified values will be less time-consuming than retrieving and restoring old values in case of rollback. Moreover, most real-time processes tend to be constrained by deadlines and access to resources, rather than by the size of resource memory.

A combination of data flow analysis, schedulability analysis, and consideration of processor resources can enable detection of cases where both expected and worst-case performance can be improved. First, the initial state is copied into a "shadow" store. The program is then executed, modifying and storing into shadow variables. Shadow values are copied to actual locations once the execution has been committed.

In some cases, when the time spent in a call is large, and the subsequent code does not depend on values modified in the call, it may even be possible to evaluate both arms of a conditional, and choose the correct arm from which to copy shadow values, once the value of the condition is known. As an alternative, if there are idle processors, both branches can be speculatively evaluated, each on a different processor. Results of both branches are transmitted, but one will be discarded once the call returns.

Asynchronous call parallelism [93] can be extended to speculative call parallelism if values referenced after the call are only conditionally modified within the body of the call. cached values or defaults can also be used to handle some cases of

data dependence with shadow execution, simply discarding results if the data values returned are not the defaults. By analogy to conditionals, in rare cases, results for several default cases could even be computed; this has not been explored in the current system.

Shadow execution also interacts favorably with while-loop iteration: the loop body can be speculatively executed, and only commit its values once it is known that the next iteration does in fact occur. With enough excess memory, an arbitrary number of future iterations could even be speculatively executed, each using the data generated by the previous iteration, and writing to a distinct copy of possibly-modified variables; once the number of iterations is known, simply the values are copied from the last iteration which actually occurs. Intuitively, the loop is unrolled and all instances of possibly-modified variables are replaced by write-once variables (although actually the same variable instance can have multiple writes within the loop body). Thus, the worst-case execution can even be enhanced.

In the next subsection, this approach is illustrated by an example to show how to perform the transformation.

4.4.4 An Example of Code Transformation

Considering the code fragment of *Figure* 4.5, suppose that none of `stmt_1` through `stmt_k` uses the parameters `x` and `y` of the method call `m(x,y)`. Then these statements can be speculatively executed concurrently with the call to method `m`. The speculative `fork` construct causes `m(x,y)` to be evaluated concurrently with block `s` containing `stmt_1` through `stmt_k`. The block containing `m`, is called the *master block*; all other forked blocks are called *slave blocks*. (In this example, there is only one slave block, the one containing `stmt_1` through `stmt_k`.)

ORIGINAL

```
m(x,y);
if (y > 1) then
  <stmt_1>;
  ...
  ...
  <stmt_k>:
  stmt_k+1: z:=x+100;
  ...
  ...
  <stmt_n>;
else
  <else_seq>
end if;
```

TRANSFORMED

```
fork
  <stmt_1>;
  ...
  ...
  <stmt_k>:
end fork;
m(x,y);
t = (y > 1)
if (t) then
  commit results
  stmt_k+1: z:=x+100;
  ...
  ...
  <stmt_n>;
else
  ignore results
  <else_seq>
end if;
```

Figure 4.5 Code transformation for speculative execution

The slave block s writes only on a local *shadow memory* space specific to s. If s itself calls a method, then the execution of that method is itself speculative, and the processor on which the called method runs must also write to shadow memory until the method commits. In the above example, if `y > 1` is true after evaluating `m(x,y)`, then the effects of statements `stmt_1` through `stmt_k` are asserted globally; if `y > 1` is false, then only the effect of `m(x,y)` is asserted. One special case should be avoided, namely, if the speculative calls eventually come back to the original processor. Consider the following scenario: process A has some statements running speculatively on a different processor; one of those statements is a call to a method assigned on the same processor on which A is running; this call either preempts or waits for the speculative code. In such a situation, the speculative call should not be made. The program call graph can be consulted to safely detect this case.

Note that the `fork` construct can be generalized to an arbitrary multiway fork, generalizing the two-way example above. All slave blocks whose condition variables evaluate to `true` are asserted, in sequence, to commit the execution. Such a construct is useful if there might be time to complete more than one slave block while the remote method is executing. In the above example, it is possible to fork `stmt_1` and `stmt_2` while forking statements `stmt_3` through `stmt_k` on a different processor. Moreover, statements for the else clause can be handled similarly, committing only appropriate blocks according to the condition. A clear consideration of doing multiway fork is the profitability of the transformation, that is, the overhead compared to the gain in performance.

This example illustrates the transformation but raises the issue of safety of the transformation.

4.5 Issues of Speculative Execution for Real-Time Systems

In this section, issues related to applying speculative execution to real-time systems are addressed. The discussion starts with safety issues concerning timeliness and data flow dependence, followed by possible interaction with real-time optimization techniques.

4.5.1 Ensuring Timeliness

Conventional program optimizations, such as those that improve a program's concurrency, may be viewed as transformations on the intermediate code of the program. From the example in *Section* 4.4, speculative execution can be considered as a program transformation which enhances a program's concurrency. The important issue is whether the transformation for speculative execution ensure timeliness.

It is possible, though not guaranteed, that transformations for speculative execution will improve a program's deadline satisfaction. However, a poorly-chosen transformation may make it difficult to satisfy deadlines. Forking requires signaling all participating processors to start their blocks; let us assume a forking time of t_f. When the fork ends, the master block's processor P_m must signal the processor on which the successful slave is running (P_s) as to whether the slave has succeeded or failed (generally the cheaper case); P_s must then transmit an update of the memory space to all affected processors, as a broadcast or a series of separate messages. The time for these "joining" communications is t_j. Note that both t_f and t_j depend on the assignment of processors and communication links to the blocks. Issues like the network topology, communication medium and communication protocols can be important factors, especially if the execution time of the slave blocks is small.

The deadlines after the `fork` construct are preserved only when the sum of the time of the master block plus that of the longest slave block (worst-case execution

path), is at least the time for *all* slave blocks to execute, plus the fork and join time. Let t_{max} be the longest execution time of any set of slave blocks which are executed under the same conditions. Then,

$$t_m + t_{max} \geq \max(t_m, t_{s_1}, t_{s_2}, ...) + t_f + t_j.$$

Therefore, the following condition must be satisfied to guarantee timeliness:

$$\min(t_m, t_{max}) \geq t_f + t_j.$$

4.5.2 Ensuring Correct Semantics

Timeliness is not the only property to be preserved during transformation. Data dependence has to be considered as well. Data dependence can be an obstacle for speculative execution, although, as indicated, default or current values may be used (although this possibility is not considered further in this thesis). Other constructs may also inhibit speculative execution. Changing a pointer variable or freeing a structure can cause problems, particularly when it involves access to live memory (in contrast to new allocation or region temporaries). In general, it is not possible to afford to checkpoint memory reached from an arbitrary live variable; however, shadow memory can be used, disposing the copy only after the speculative execution commits. For example, if a pointer is to be dereferenced, a shadow copy pointing to the same structure can be created and the original pointer can be set free. If the execution is committed, the shadow copy is dereferenced. Otherwise the pointer is restored from the shadow copy.

Writing to output or reading from input causes other difficulties. Writing can often be handled by intercepting the writes in a private buffer, and writing on commit. The buffer cannot be shared by multiple processes to avoid producing invalid semantics. The same approach can, with greater difficulty, be used for input. The values read from input can be buffered and provided to their eventual targets,

provided that no other process is concurrently reading the same input file, and that the data would eventually be read elsewhere (to avoid invalid "file empty" errors).

Other interactions with the external environment, and certain types of interactions with resource managers, must also be avoided, e.g., disconnection of a channel, destruction of a socket, or a font-change message to a printer. Finally, some exceptions have persistent effects on the environment; these effects have to be intercepted and buffered for possible later commit.

Because of these and other issues, speculative execute is enabled only when the code to be speculatively parallelized provably contains no unsafe construct, such as those discussed above; future work includes identification of unsafe constructs and of cases in which speculative execution can proceed in their presence, and the modifications and transformations needed to assure the safety of doing so.

4.5.3 Interaction with Real-Time Optimization Techniques

Some optimization techniques for real-time systems can interact positively with speculative execution. For example, an optimization technique may enable more opportunities for speculative execution, or allow further optimization in the speculatively executed code.

Given a computation, for instance $x := sin(y) * y + c()$, with data dependence on the value of a variable x, a value c_0 for $c()$ may be assumed and later the value(s) affected by that assumption can be updated. When the true value c_1 is returned, simply $c_1 - c_0$ can be added to x, and other values depending on x can be adjusted. The code added to make that adjustment is called "Δ-code" [76, 77]. If the dependence on the return value is "simple", so that Δ-code can be easily specified, and the time to return from the callee is greater than the time for the

Δ-code update, the code can speculatively evaluated for the current value (or some guessed or default value), and use the Δ-code to adjust the solution.

Another situation in which speculative execution can interact positively with optimization techniques is when the callee method does not modify its own (or transitively, its descendants') state, and does not produce observables. In that case, parameter patterns can *cache* and values can be returned, and return values can be reused instead of making a call (provided that testing of equality for parameter lists is cheap) [82]. If tests are not cheap, but *kill* message can be passed, a call may be started, and the initial state can be stored at the callee. If later the call is found to be unnecessary, a kill message can be sent. In some cases, it may be required to bypass a second call even before the first call has returned. This guarantee of a state not being modified must be a user assertion or a compile-time guarantee, perhaps by data-flow analysis, and it is complicated by pointers, complex array index expressions, or structures [54]. These techniques are not explored further in this paper.

In the next section, the compiler transformation rules for speculative execution are specified. While considering issues discussed in this section, the rules require a satisfaction of a set of preconditions to avoid violating the program timeliness or changing the program semantics.

4.6 Compiler Transformations for Speculative Execution

After illustrating various issues concerning speculative execution, in this section, the specification of compiler transformation rules for speculative execution is presented, *Figures* 4.6 to 4.7. The rules detect opportunities for speculative execution in conditional statements and while loops. In addition, a dual set of rules, addressing possible chances for shadow execution, are included in *Appendix C.*

RULE:	SPECULATIVE_IF
Preconditions:	Structural:
	(1) S = (if (C) $then$ S_2 $else$ S_3) is a single-exit code region.
	(2) C is a call being executed on another processor
	Dependence:
	(3) $Vars(S_2) \cap Mod(C) = \phi$
	(S_2's variables have correct values immediately before if)
	Blocking:
	(4) There are no blocking constructs in S_2.
	(5) For all methods M in $TCalls(Calls(S_2))$, $not(Blocking(M))$.
	(6) For each method M in $TCalls(C) \cap TCalls(Calls(S_2))$, $not(Ordered(M))$.
	(Incorrectly or prematurely executing any such statement has a permanent and invalid effect on the environment.)
	Timing:
	(7) $t_s(Mod(S_2)) + t_f + t_j < Time(C)$.
	(Useful work can be done.)
	(8) $Time(S_3) + t_r(Mod(S_2)) \leq Time(S_2)$.
	(Worst-case time does not increase.)
Actions:	
	Execute C in parallel with the following:
	$save(Mod(S_2))$; S_2.
	Insert synchronization between $exit(C)$ and $exit(S_2)$.
	Check x_c, the return parameters of C;
	If this enables S_2, do nothing.
	Otherwise, execute $restore(Mod(S_2))$; S_3.
	In any case, continue executing from $exit(S)$.
Postcondition:	
	S has completed without missing its deadline.
	State is as if execution had been sequential.
Comment:	
	A symmetric rule exists for S_3.
Properties:	
	Preserves the program semantics.
	Does not extend the worst-case execution path.

Figure 4.6 Speculative execution for if clauses

In the Speculative-If rule (*Figure* 4.6), the condition of the if-statement is assumed to include a call that can be executed on another processor. One of the branches can be selected to be speculatively executed while making the call C in the condition. The selected branch (S_2 for example) should satisfy the following conditions :

1. The variable used in S_2 are not modified as a side effect of the call C.
2. Neither S_2 nor any function transitively called from S_2 has a blocking construct.
3. There will not be a change in the order of any calls to critical sections if S_2 runs while C is running.

To ensure the safety and profitability of the transformations, timing conditions of (7) and (8) in *Figure* 4.6 must be satisfied before performing the transformations. Safety can be guaranteed if the worst-case execution path is not extended. Speculatively executing the longest branch of a condition, the effect of rollback on the other branch should be examined. The rollback penalty should not extend the short branch over the worst-case execution time, as in (8). The transformation is profitable if the overhead of storage, forking and joining is less than the execution time of C, as described in (7).

The SPECULATIVE-WHILE rule, in *Figure* 4.7, follows in the same spirit. The shadow execution rules are based on the same preconditions considering copy-and-commit rather than rollback. A full list of the rules for speculative execution and shadow execution can be found in *Appendix C.*

We have targeted various real-time image processing techniques to investigate applicability of speculative execution [115]. A significant potential has been found for safe and profitable speculative execution in image compression, edge detection, morphological filters, and blob recognition. In addition, speculative execution can

RULE:	SPECULATIVE_WHILE
Preconditions:	Structural:
	(1) S = (*while* (C) *do* S_2) is a single-exit code region.
	(2) C is a call being executed on another processor
	(3) The loop will be executed at least once.
	Dependence:
	(4) $Vars(S_2) \cap Mod(C) = \phi$
	(S_2's variables have correct values immediately before *while*)
	Blocking:
	(5) There are no blocking constructs in S_2.
	(6) For all methods M in $TCalls(Calls(S_2))$, $not(Blocking(M))$.
	(7) For each method M in $TCalls(C) \cap TCalls(Calls(S_2))$, $not(Ordered(M))$.
	Timing:
	(8) $t_r(Mod(S_2)) + t_s(Mod(S_2)) + t_f + t_j < Time(C)$.
	(Useful work can be done; worst-case time does not increase.)
	(9) $t_r(Mod(S_2)) \leq Time(S_2)$.
	(Given at least one iteration; worst-case time does not increase.)
Actions:	
	Execute C in parallel with the following:
	$save(Mod(S_2))$; S_2.
	Insert synchronization between $exit(C)$ and $exit(S_2)$.
	Check x_c, the return parameters of C;
	If this enables S_2, repeat.
	Otherwise, execute $restore(Mod(S_2)); exit(S)$.
Postcondition:	
	S has completed without missing its deadline.
	State is as if execution had been sequential.
Properties:	
	Preserves the program semantics.
	Does not extend the worst-case execution path.

Figure 4.7 Speculative execution for while clauses

improve other properties of real-time systems, such as fault tolerance [117]. The next three chapters concentrate on the compiler rules for speculative execution. First, a formal proof is provided for safety of those rules. Then, empirical validation efforts are described emphasizing the contribution of speculative execution to performance and timeliness of real-time systems. A study based on simulation for effects of various characteristic of real-time programs on the applicability and usefulness of speculative execution is presented in *Chapter* 6. *Chapter* 7 provides a more aggressive validation for speculative execution in realistic applications through prototyping.

CHAPTER 5

FORMAL VERIFICATION OF SPECULATIVE EXECUTION RULES

The previous chapter presented compiler transformation rules for speculative execution. A global requirement for all compiler transformations is to preserve program semantics. This is a safety requirement. For real-time programs, safety has a more restrictive definition: code transformations should also not worsen the timing properties of the program; a program that meets all timing constraints should not be transformed to a one that fails its deadline. Thus, it is necessary to verify that speculative execution does preserve the program semantics and timeliness. This is especially important when applied to safety-critical real-time applications, such as patient monitoring, avionics and air-traffic control, where errors may be disastrous. In this chapter, formal verification of semantic correctness and timeliness for the speculative execution transformation rules is provided.

A correctness proof of a non-real-time compiler transformation consists of three parts: first, using a technique such as abstract interpretation [26] to show that the data flow equations correctly abstract the program semantics; second, proving that the data flow computation terminates; third, proving that the transformations preserve the semantics. For real-time systems, there are three corresponding proofs regarding timing: the correctness of individual timing rules, the correctness of timing summaries, and the preservation of desired timing properties by the transformation. Assuming that the first two proofs are given, i.e., given correct data flow and timing information, we show that the transformation preserves the required properties, *program semantics* and *timeliness* [69, 73, 79].

In reasoning about programs, there are two types of proof systems, exogenous and endogenous. The assertions of an exogenous logic [43, 74] such as "$P\{S\}Q$"

contain program fragments (S) and assert the fragments using precondition P and postcondition Q. With exogenous logics, there is one axiom for each programming language construct, which makes it suitable for statement-based transformations. On the other hand, endogenous logic [6, 40, 75] does not consider intermediate states and thus is suitable for block-based transformations including this work.

Research in [3, 10] investigates preservation of the meaning of programs, in the context of a functional language. But they do not handle timeliness, the most important property of real-time systems. On the other hand, it has been known that methods developed for shared-memory multiprocessing apply equally well to distributed systems and real-time systems (with an additional variable *time*) [1], so the temporal logics of concurrent systems [6, 15, 42] can be applied to real-time programs. The goal is to prove that if a property holds originally in a real-time program, it will hold after applying transformation rules. The focus is on verifying the semantic correctness and timeliness of a program. We don't reason step-by-step on the statements between P and Q; hence, an endogenous logic based on the temporal logic of Owicki and Lamport [40, 75] has been adopted.

In this chapter, verification of the speculative execution compiler transformation rules is provided. Based on the discussion above, verification of real-time compiler transformations must address two properties: preserving semantics and timeliness. In particular, we verify transformation rules for speculative execution of conditional branches and while loop bodies on the same processor which were shown in *Figures* 4.6 and 4.7. Symmetric proofs exist for shadow execution rules.

5.1 Semantic Correctness Proof

In this section the transformation rules are shown to preserve the semantics of a program. The goal is to prove that applying a transformation rule should lead to

a semantically equivalent state, according to the definition of semantic equivalence below.

In *Figure* 1.4, let $\sigma_{code\ block1}$ and $\sigma_{code\ block2}$ denote the states, with no speculative execution, after the execution of *code block1* and *code block2* respectively. Suppose after applying the transformation rules $\sigma'_{code\ block1}$ and $\sigma'_{code\ block2}$ denote the states after the execution of *code block1* and *code block2* respectively. The transformation preserves the program semantics if

1. the states, $\sigma_{code\ block1}$ and $\sigma'_{code\ block1}$ are semantically equivalent, and
2. $\sigma_{code\ block2}$ and $\sigma'_{code\ block2}$ are semantically equivalent.

In other words, it is necessary to show that the extra computation (e.g., fork or copy) incurred by the transformation, as indicated in *Figure* 1.4, leads to a state which is semantically equivalent to the state without the extra computation.

Throughout this chapter, S is used to denote a code segment, and S^0 is the corresponding transformed code of S. The following notations and definitions are necessary for the proof.

Notation 5.1.1 *Let Σ denote the set of states of a program P. Let σ_S denote the state of a program after executing the code segment S, where $\sigma_S \in \Sigma$.*

Notation 5.1.2 *Let $\sigma(x)$ be the value of variable x at the state σ.*

Note that x may not be initialized or declared in σ. These two cases are handled separately in the semantic correctness proof.

Notation 5.1.3 *Let Π be the projection of states onto the set of variables live immediately after the execution of S (i.e., after(S)), and Π' be the corresponding projection immediately after the execution of S^0 (i.e., $after(S^0)$).*

Definition 5.1.1 *Given σ_S and σ_{S^0}, we say that the states σ_S and σ_{S^0} are equivalent, denoted by $\sigma_S = \sigma_{S^0}$, if for every $x \in \Pi\sigma_S$ and $x \in \Pi\sigma_{S^0}$, the value of x is the same in σ_S and σ_{S^0}.*

There is a subtlety in the above notation, namely, the use of projections onto the set of variables *live* on exit of the transformed code block. Many compiler transformations for optimization and parallelization, and some for speculative execution, either introduce new variables, or eliminate unnecessary variables, which result in different values for variables, where, in each case, those values are never used after exiting the transformed block. Clearly, such *dead* values should not matter in evaluating the correctness of the transformation. As far as correctness concerns, if a variable exists before transformation, its value should be the same after applying transformation rules. Projections are used to show that for the variables of interest, values are not altered after transformation. We actually use a weaker criterion, differing when the initial program does not terminate correctly.

Definition 5.1.2 *Given σ_S and σ_{S^0}, we say that the semantic of a program is (weakly) preserved after transformation if and only if the following two conditions hold:*

1. *if S converges and $\sigma_S \in \Sigma$, then $\sigma_{S^0} \in \Sigma$, and $\Pi\sigma_S = \Pi'\sigma_{S^0}$.*
2. *if S converges but $\sigma_S \notin \Sigma$, (that is, σ_S is an error state), then S^0 also converges.*

The above definition of semantic equivalence is motivated by a requirement that any transformation should not worsen the quality of the results returned by a program. If the computation terminates normally and returns output (correct by assumption), the transformed program must return the same output. If, however, the program terminates in error, then we certainly should not object if the transformation eliminates the error and returns a correct result; arguably if less clearly,

we shouldn't be particularly concerned once an error has occurred, as long as the program terminates. Finally, if a program does not terminate, any behavior in the transformed program is acceptable. Note that this criterion can be modified to prohibit catastrophic errors in the transformed programs, perhaps by adding an ordering on errors, and requiring the error in the transformed program to be more severe.

Semantic correctness is preserved if a transformed program preserves the contents of the program store during execution. So the program store corresponding to each program state is examined to show semantics-preservation, as in denotational semantics [3] or abstract interpretation [26].

In the semantic proof, both fork and join are assumed to have no side effects on the program state, since they usually represent only operating system overhead to manage the new process.

Lemma 5.1.1 *The semantic of a program after applying the speculative if-rule, Figure 4.6, is preserved assuming the computation converges.*

Proof: Let the following equations denote, respectively, the state sequences before and after applying the transformation rule.

$$\cdots \xrightarrow{S_a} \sigma_P \xrightarrow{S_b} \sigma_Q \xrightarrow{S_c} \cdots$$

$$\cdots \xrightarrow{S_a} \sigma_P \xrightarrow{S_b^0} \sigma_{Q^0} \xrightarrow{S_c} \cdots$$

where S_b denotes the statement $if\ (C)\ then\ S_2\ else\ S_3$, and $S_b^0 = (x_c = C \parallel (Save; Fork; S_2));\ if\ (\neg x_c)\ then\ Restore; S_3$, is the corresponding transformed version. It is required to show that for an arbitrary variable x, the value of x will be the same in both σ_Q and σ_{Q^0}. Therefore, the states σ_Q and σ_{Q^0} are semantically equivalent.

The value of x may be modified in the transformed version as follows:

1. x_c is used only locally in the evaluation of C, and is not live on exit from S_b, so is unconstrained.

2. If x is not a live variable when the control reaches C, there are no constraints on $\sigma_Q(x)$ or $\sigma_{Q^0}(x)$. Thus x is assumes henceforward to be a live variable upon entering the execution of C.

3. $x \in Pres(S_b) \Longrightarrow x \in Pres(S_b^0)$

   $\Longrightarrow \sigma_P(x) = \sigma_Q(x) = \sigma_{Q^0}(x)$

   If x is never modified in S_b then its value will be the same in σ_{Q^0} since the operations incurred by the transformation such as fork and save do not modify the value. Thus, $\sigma_Q(x) = \sigma_{Q^0}(x)$.

4. $x \in Mod(S_2)$ and $x \notin Mod(C) \cap Mod(S_3)$

   C: $\Longrightarrow S_2$ is to be executed.

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C\|(Save;Fork;S_2)}(x) = \sigma_{S_2}(x)$ (Since $x \notin Mod(C)$)

   $\Longrightarrow \sigma_Q(x) = \sigma_{C;S_2}(x) = \sigma_{S_2}(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

   $\neg C$: $\Longrightarrow S_3$ is executed.

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C\|(Save;Fork;S_2);Restore;S_3}(x) = \sigma_{Restore;S_3}(x)$

   $\Longrightarrow \sigma_{Restore;S_3}(x) = \sigma_{Restore}(x)$ (Since $x \notin Mod(S_3)$)

   $\Longrightarrow \sigma_{Restore}(x) = \sigma_P(x)$ (Since $x \notin Mod(C)$)

   $\Longrightarrow \sigma_Q(x) = \sigma_{C;S_3}(x) = \sigma_P(x)$ (Since $x \notin Mod(S_3)$ and $x \notin Mod(C)$)

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

   If the then-clause is to be executed, the effect of S_2 on the value of x will be propagated to σ_{Q^0}. However, since x is not modified by C, restoring

the value of x of σ_P is sufficient to preserve the semantics when following the else branch.

5. $x \in Mod(S_3)$ and $x \notin Mod(S_2) \cap Mod(C)$

   C: $\sigma_{Q_0}(x) = \sigma_Q(x)$ since x is not modified in S_2.

   $\neg C$: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C\|(Save;Fork;S_2);Restore;S_3}(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{S_3}(x)$ (Since $x \notin Mod(S_2) \cap Mod(C)$)

   $\Longrightarrow \sigma_Q(x) = \sigma_{C;S_3}(x) = \sigma_{S_3}(x)$ (Since $x \notin Mod(C)$)

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

   Since x is not modified by either C or S_2, the speculative execution of the then-clause has no effect on the value of x. Therefore, the semantic is preserved if the condition C is true. On the other hand, when the else-clause is executed, the value of x in σ_P is used, which leads to the same state as the serial order of execution.

6. $x \in Mod(C)$ and $x \notin Mod(S_2) \cap Mod(S_3)$

   $\Longrightarrow \sigma_Q(x) = \sigma_{(C;S_2)\vee(C;S_3)}(x) = \sigma_C(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{(C\|(Save;Fork;S_2))\vee(C\|(Save;Fork;S_2);Restore;S_3)}(x)$

   Since the value of x is not modified in either branch, only the effect of C on x will be propagated to σ_Q.

   C: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{(C\|(Save;Fork;S_2))}(x) = \sigma_C(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

   $\neg C$: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{(C\|(Save;Fork;S_2);Restore;S_3)}(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{(C;Restore;S_3)}(x) = \sigma_{C;S_3}(x)$ (Since $x \notin Mod(S_2)$)

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C;S_3}(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

In case of rollback, the value of x will not be restored to $\sigma_P(x)$. As the values of variables modified in S_2 are only restored, the modification of x in C will not be overwritten by rolling back execution.

7. $x \in Mod(S_2) \cap Mod(S_3)$ and $x \notin Mod(C)$

C: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C\|(Save;Fork;S_2)}(x) = \sigma_{S_2}(x)$

There is no race condition due to the parallel execution of C and S_2, since x is not modified in C

$\Longrightarrow \sigma_Q(x) = \sigma_{C;S_2}(x) = \sigma_{S_2}(x)$

$\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

$\neg C$: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{(C\|(Save;Fork;S_2);Restore;S_3)}(x)$

$\Longrightarrow \sigma_{Q^0}(x) = \sigma_{S_2;Restore;S_3}(x)$ (Since $x \notin Mod(C)$)

$\Longrightarrow \sigma_{Q^0}(x) = \sigma_{S_2;Restore;S_3}(x) = \sigma_{S_3}(x)$

As $x \in Mod(S_2)$, the value of x will be restored to its original value in σ_P

$\Longrightarrow \sigma_Q(x) = \sigma_{C;S_3}(x) = \sigma_{S_3}(x)$ (Since $x \notin Mod(C)$)

$\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

8. $x \in Mod(C) \cap Mod(S_3)$ and $x \notin Mod(S_2)$

C: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C\|(Save;Fork;S_2)}(x) = \sigma_C(x)$ (Since $x \notin Mod(S_2)$)

$\Longrightarrow \sigma_Q(x) = \sigma_{C;S_2}(x) = \sigma_C(x)$

$\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

$\neg C$: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{(C\|(Save;Fork;S_2);Restore;S_3)}(x)$

$\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C;Restore;S_3}(x)$ (Since $x \notin Mod(S_2)$)

$\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C;Restore;S_3}(x) = \sigma_{C;S_3}(x)$

As $x \notin Mod(S_2)$, the value of x does not need to be restored to its original value in σ_P

$$\Longrightarrow \sigma_Q(x) = \sigma_{C;S_3}(x)$$

$$\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$$

9. $x \in Mod(C) \cap Mod(S_2)$ cannot occur, by construction.

10. $x \in Mod(C) \cap Mod(S_2) \cap Mod(S_3)$ cannot occur, by construction.

Lemma 5.1.2 *The semantic of a program after the speculative execution of while-rule, Figure 4.7, is preserved assuming the computation converges.*

Proof: For while loops, semantic safety will follow if the states after each iteration (meaning, for the original, after the execution of S_2, and for the transformed code, at synchronization) are shown to be equivalent. Also the numbers of committed executions of S_2 are the same. It is sufficient to show (1) that a single execution of the loop transforms the value of a variable x identically if C holds, and (2) that, when C fails, $x's$ value is likewise transformed identically.

Recall that, $S_b = While\ (C)\ do\ S_2$, is transformed into $S_b^0 = While\ (x_c = C \parallel (Save; Fork; S_2))$; *Restore*. Assume that the state sequence of the original program is $\cdots \xrightarrow{S_a} \sigma_P \xrightarrow{S_b} \sigma_Q \xrightarrow{S_c} \cdots$ and after applying the speculative while-rule of *Figure* 4.7, the state sequence becomes $\cdots \xrightarrow{S_a} \sigma_P \xrightarrow{S_b^0} \sigma_{Q_0} \xrightarrow{S_c} \cdots$. It is required to prove that $\sigma_Q = \sigma_{Q^0}$.

Assume that the while loop is executed n times, as well as I_i and E_i are the states before and after executing the i^{th} iteration, respectively. The original state sequence can be expanded further into:

$$\cdots \xrightarrow{S_a} \sigma_P\ \sigma_{I_1} \xrightarrow{C;S_2} \sigma_{E_1}\ \sigma_{I_2} \xrightarrow{C;S_2} \sigma_{E_2} \cdots \sigma_{I_n} \xrightarrow{C;S_2} \sigma_{E_n}\ \sigma_{I_{n+1}} \xrightarrow{C} \sigma_Q \xrightarrow{S_c} \cdots$$

where $\sigma_P = \sigma_{I_1}$, and $\sigma_{I_{i+1}} = \sigma_{E_i}$, $\quad i = 1, \cdots, n$ and the corresponding expanded state sequence after the transformation is

$$\cdots \xrightarrow{S_a} \sigma_P\ \sigma_{I_1^0} \xrightarrow{C \parallel (Save;Fork;S_2)} \sigma_{E_1^0}\ \sigma_{I_2^0} \xrightarrow{C \parallel (Save;Fork;S_2)} \sigma_{E_2^0} \cdots$$

$$\sigma_{I_n^0} \xrightarrow{C\|(Save;Fork;S_2)} \sigma_{E_n^0}\ \sigma_{I_{n+1}^0} \xrightarrow{C\|(Save;Fork;S_2);Restore} \sigma_{Q^0} \xrightarrow{S_5} \cdots$$

where I_i^0 and E_i^0 are the corresponding states to I^0 and E^0, respectively, in the transformed code, and $\sigma_P = \sigma_{I_1^0}$, and $\sigma_{I_{i+1}^0} = \sigma_{E_i^0}$, $i = 1, \cdots, n$

As for speculative-if, it is sufficient to show that the value of an arbitrary variable x will be the same in both σ_Q and σ_{Q^0}. Therefore, it is necessary to consider all the possible cases of propagating the value of x to the σ_{Q^0}:

1. Since x_c is used only locally in the evaluation of C, it is neither live from iteration to iteration nor upon exit from S_b, and therefore it is unconstrained.

2. $x \in Pres(S_b) \Longrightarrow x \in Pres(S_b^0)$

   $\Longrightarrow \sigma_P(x) = \sigma_A(x) = \sigma_{A^0}(x) = \sigma_Q(x) = \sigma_{Q^0}(x)$

   If x is not modified in S_b, the value of x will be the same for all program points A in the execution of S_b.

3. $x \in Mod(S_2)$ and $x \notin Mod(C)$

   C: assume that $\sigma_{I_i}(x) = \sigma_{I_i^0}(x)$, $i = 1, \cdots, n$

   $\Longrightarrow \sigma_{E_i^0}(x) = \sigma_{after(C\|(Save;Fork;S_2))}(x) = \sigma_{S_2}(x) = \sigma_{E_i}(x)$

   Since $\sigma_{I_1}(x) = \sigma_{I_1^0}(x) = \sigma_P(x)$

   $\Longrightarrow$ by induction $\sigma_{E_i^0}(x) = \sigma_{E_i}(x)$ $\forall i = 1, \cdots, n$

   $\neg C$: $\Longrightarrow \sigma_Q(x) = \sigma_C(x) = \sigma_{I_{n+1}}(x) = \sigma_{E_n}(x)$

   $\Longrightarrow \sigma_Q(x) = \sigma_{E_n}(x) = \sigma_{E_n^0}(x)$ (just proven by induction)

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{after(C\|(Save;Fork;S_2);Restore)}(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{after(Save;Fork;S_2;Restore)}(x) = \sigma_{I_{n+1}^0}(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{I_{n+1}^0}(x) = \sigma_{E_n^0}(x)$

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

If x is not modified in C, the value of x after every iteration will be the result of the operation in S_2, similar to the original code. However, on exiting the loop, the value of x after the last iteration needs to be restored.

4. $x \in Mod(C)$ and $x \notin Mod(S_2)$

   C: assume that $\sigma_{I_i}(x) = \sigma_{I_i^0}(x), \quad i = 1, \cdots, n$

   $\Longrightarrow \sigma_{E_i}(x) = \sigma_{C;S_2}(x) = \sigma_C(x)$

   $\Longrightarrow \sigma_{E_i^0}(x) = \sigma_{C\|(Save;Fork;S_2)}(x) = \sigma_C(x)$

   $\Longrightarrow \sigma_{E_i^0}(x) = \sigma_{E_i}(x)$

   Since $\sigma_{I_1}(x) = \sigma_{I_1^0}(x) = \sigma_P(x)$

   $\Longrightarrow$ by induction $\sigma_{E_i^0}(x) = \sigma_{E_i}(x) \quad \forall i = 1, \cdots, n$

   $\neg C$: $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{(C\|(Save;Fork;S_2);Restore)}(x) = \sigma_{C;Restore}(x)$

   Given that $Mod(S_2)$ is only restored

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_{C;Restore}(x) = \sigma_C(x)$

   $\Longrightarrow \sigma_Q(x) = \sigma_C(x)$

   Given that $\sigma_{I_{n+1}^0}(x) = \sigma_{E_n^0}(x) = \sigma_{E_n}(x) = \sigma_{I_{n+1}}(x)$

   (just proved by induction)

   $\Longrightarrow \sigma_{Q^0}(x) = \sigma_Q(x)$

   As the value of x is affected only by the execution of C, $\sigma_{E_i^0}(x)$ after any iteration i is similar to the original code.

5. $x \in Mod(C) \cap Mod(S_2)$ cannot occur, by construction.

Theorem 5.1.1 *Given the transformation rules of Figure 4.6 and Figure 4.7, the (weak) semantic of a program is preserved after applying the if-rule and while-rule.*

Proof: Let Σ denote the set of states of a program P. Let S denote an *if* or a *while* statement which satisfies the constraints of Figure 4.6 or Figure 4.7. Let

S^0 be the corresponding transformed code of S. It is required to show that the semantic of the program P is preserved after transformation regardless of divergence or convergence of S. Let Π and Π' be projections on live variables at $exit(S)$ and $exit(S^0)$ respectively. Also, let σ_S denote the state after the execution of S. There are three cases to consider.

Case 1: S converges and $\sigma_S \in \Sigma$. From Lemma 1 and Lemma 2, S^0 converges, the state $\sigma_{S^0} \in \Sigma$, and $\Pi\sigma_S = \Pi'\sigma_{S^0}$.

Case 2: S converges and $\sigma_S \notin \Sigma$, i.e., the execution of S leads to an erroneous state. Since the extra computation incurred by the transformation rule, such as fork and save, is finite, S^0 will terminate.

Case 3: S diverges. No guarantee is required on termination of S^0 or the value of σ_{S^0} after the execution of the transformed code S^0.

From Definition 5.1.2, the weak semantic of a program is preserved after the transformation. □

5.1.1 Timeliness Proof

Timeliness should be guaranteed before a transformation is performed. Consider the example of *Figure* 1.4. If the execution time of each block is as defined as in *Section* 1.4, the timing preconditions of the rules should prevent transforming the code as the worst-case time is extended (it becomes 19 units instead of 18 units in the original code). In this subsection, the issue of the timing property of the transformations is addressed. In order to prove safety of a transformation rule, it is necessary to verify that the worst-case execution time is not increased. In the following proof, WCT, □, $\rightsquigarrow$ are used to denote worst-case time, always true, and leads to, respectively. In addition, the notation $Path(P, T, Q)$ means that Q is reachable from P through T. As for the semantic correctness proof, we begin by showing that timeliness is preserved for the speculative-if rule.

Theorem 5.1.2 *The speculative-if rule of* Figure *4.6 does not extend the worst-case execution of the code after transformation.*

Proof: Assume that a program meets its deadline, and has the property $P \rightsquigarrow Q$ for the code segment $if\ (C)\ then\ S_2\ else\ S_3$ where P denotes $at(if)$ and Q is $after(if)$. The following holds according to the constraints of *Figure* 4.6.

- $\Box P \Longrightarrow Path(P,R,Q) \vee Path(P,H,Q)$, using the proof system in [40], where R and H represent the state formulas following the execution of *then* and *else* branches, respectively.
- $P \overset{\leq WCT}{\rightsquigarrow} Q$, (given that the program meets its deadline).
- $R \overset{\leq Time(S_2)}{\rightsquigarrow} Q$
- $H \overset{\leq Time(S_3)}{\rightsquigarrow} Q$
- $P \overset{\leq Time(C)}{\rightsquigarrow} R \vee H$
- $Time(C) + Time(S_2) \leq WCT$, (assuming that the then-clause is the longer branch).
- $Time(C) + Time(S_3) < WCT$

After applying the speculative-if rule, the program structure is modified, and the path of reaching Q from P is different.

Assume that $\Box P \Longrightarrow Path(P,R^0,Q) \vee Path(P,H^0,Q)$, where R^0 and H^0 represent the state formulas denoting $at(S_2)$ and $at(S_3)$ respectively.

$\Longrightarrow \left(P \overset{\leq Time(C)}{\rightsquigarrow} R^0 \vee H^0\right)$

With speculative execution, R^0 or H^0 are reachable from P after the execution of C. Thus Q may be reached from them at different time. C is executed in parallel with forking a new process for executing S_2 after saving the original state. It is required

to rollback before executing S_3.

$\Longrightarrow \left(H^0 \overset{\leq Time(S_3)+t_r}{\rightsquigarrow} Q\right)$

$\Longrightarrow \left(R^0 \overset{\leq Max((Time(S_2)+t_s+t_j+t_f),Time(C))}{\rightsquigarrow} Q\right)$

Therefore, the time to reach Q from P depends on the path, and is less than the execution time of longest branch.

$\Longrightarrow \left(P \overset{\leq Max((Time(S_3)+t_r+Time(C)),(Time(S_2)+t_s+t_j+t_f),Time(C))}{\rightsquigarrow} Q\right)$

Using the timing preconditions of the transformation rule of *Figure* 4.6, we now prove that the worst-execution time is not extented after transforming the code.

If $Max((Time(S_2)+t_s+t_j+t_f), Time(C)) = Time(C)$

$\Longrightarrow \left(P \overset{\leq Time(C)+Time(S_3)+t_r}{\rightsquigarrow} Q\right)$

$\Longrightarrow \left(P \overset{\leq Time(S_2)+Time(C)}{\rightsquigarrow} Q\right)$ (Using (8) of Figure 4.6).

$\Longrightarrow \left(P \overset{\leq WCT}{\rightsquigarrow} Q\right)$

If $Max((Time(S_2)+t_s+t_j+t_f), Time(C)) = Time(S_2)+t_s+t_j+t_f$

$\Longrightarrow \left(P \overset{\leq Max((Time(S_3)+t_r+Time(C)),(Time(S_2)+t_s+t_j+t_f))}{\rightsquigarrow} Q\right)$

$\Longrightarrow \left(P \overset{\leq Max(Time(S_2)+Time(C),Time(S_2)+Time(C))}{\rightsquigarrow} Q\right)$ (Using (7) of Figure 4.6).

$\Longrightarrow \left(P \overset{\leq WCT}{\rightsquigarrow} Q\right)$

Having proven the timeliness of the transformed code after applying the speculative-if rule, it is next shown that the same property holds for the while loop transformation.

Theorem 5.1.3 *The speculative-while rule of* Figure *4.7 does not extend the worst-case execution of the code after transformation.*

Proof: Similar to the case of if-statement, the original program as assumed to meet its deadline, and has the property $P \rightsquigarrow Q$ for the code segment *while* (C) *do* S_2. The following hold according to the speculative-while rule of *Figure* 4.7.

- $\Box P \Longrightarrow Path(P, I_1, E_1, I_2, E_2, ...I_n, E_n, I_{n+1}, Q)$ again using the proof system in [40], where n is the number of iterations, and I_i and E_i are the states before and after executing the i^{th} iteration, respectively.
- $P \overset{\leq WCT}{\rightsquigarrow} Q$ (given that the program meets its deadline).
- $I_i \Longrightarrow at(C)$, $i = 1, ..., n$ (every iteration starts by executing C).
- $at(C) \overset{\leq Time(C)}{\rightsquigarrow} after(C)$
- $after(C) \Longrightarrow at(S_2)$ (S_2 will be serially executed following C).
- $at(S_2) \overset{\leq Time(S_2)}{\rightsquigarrow} after(S_2)$
- $after(S_2) \Longrightarrow E_i$ $i = 1, ..., n$
- $I_{n+1} \overset{\leq Time(C)}{\rightsquigarrow} Q$
- $(n+1)Time(C) + nTime(S_2) \leq WCT$

After performing the transformation, the state transition occurs at different times. The two scenarios are considered separately. The first is when the speculative execution of S_2 is committed. The second case deals with exiting the loop.

$\Box I_i \Longrightarrow (at(C) \wedge at(Save(S_2)))$ $i = 1, \cdots, n$

(C is to be executed in parallel with S_2).

$\Longrightarrow \left(at(C) \overset{\leq Time(C)}{\rightsquigarrow} after(C)\right)$

$\Longrightarrow \left(at(Save(S_2)) \overset{\leq t_s + Time(S_2) + t_j + t_f}{\rightsquigarrow} after(S_2)\right)$

$\Box E_i \Longrightarrow (after(C) \vee after(S_2))$

$$\Longrightarrow \left(I_i \overset{\leq Max(Time(C),(t_s+Time(S_2)+t_j+t_f))}{\rightsquigarrow} E_i\right)$$

When C is false, all side effects due to the speculative execution of S_2 need to be undone.

$\Box I_{n+1} \Longrightarrow (at(C) \wedge at(Save(S_2)))$

$$\Longrightarrow \left(at(C) \overset{\leq Time(C)}{\rightsquigarrow} after(C)\right)$$

The state $after(C)$ is actually the state $at(Restore(S_2))$.

$$\Longrightarrow \left(at(Restore(S_2)) \overset{\leq t_r}{\rightsquigarrow} Q\right)$$

$$\Longrightarrow \left(I_{n+1} \overset{\leq Time(C)+t_r}{\rightsquigarrow} Q\right)$$

Considering the whole execution of the while loop, we conclude that:

$$\left(P \overset{\leq [Time(C)+t_r+nMax(Time(C),(t_s+Time(S_2)+t_j+t_f))]}{\rightsquigarrow} Q\right)$$

The next step is to compare the new worst-case execution time (due to the transformation) with the original and show that the new WCT is not greater than the original WCT. There are two cases.

If $Max(Time(C),(t_s+Time(S_2)+t_j+t_f)) = Time(C)$

$$\Longrightarrow \left(P \overset{\leq (n+1)Time(C)+t_r}{\rightsquigarrow} Q\right)$$

$$\Longrightarrow \left(P \overset{\leq (n+1)Time(C)+Time(S_2))}{\rightsquigarrow} Q\right) \quad \text{(Using (9) of Figure 4.7).}$$

$$\Longrightarrow \left(P \overset{\leq WCT}{\rightsquigarrow} Q\right)$$

Similarly, if $Max(Time(C),(t_s+Time(S_2)+t_j+t_f)) = t_s+Time(S_2)+t_j+t_f$

$$\Longrightarrow \left(P \overset{\leq Time(C)+t_r+nTime(S_2)+n(t_s+t_j+t_f)}{\rightsquigarrow} Q\right)$$

$$\Longrightarrow \left(P \overset{\leq Time(C)+nTime(S_2)+(t_r+t_s+t_j+t_f)+(n-1)(t_s+t_j+t_f)}{\rightsquigarrow} Q\right)$$

$$\Longrightarrow \left(P \overset{\leq Time(C)+nTime(S_2)+Time(C)+(n-1)Time(C)}{\rightsquigarrow} Q\right) \quad \text{(Using (8) of Figure 4.7).}$$

$$\Longrightarrow \left(P \overset{\leq (n+1)Time(C)+nTime(S_2)}{\rightsquigarrow} Q\right)$$
$$\Longrightarrow \left(P \overset{\leq WCT}{\rightsquigarrow} Q\right)$$

After formally verifying the compiler rules for speculative execution, the applicability and usefulness of speculative execution are addressed. In the next chapter, an experiment based on simulations that study the impact of various properties of real-time programs affecting the applicability of speculative execution is described. In *Chapter* 7, the results of applying the transformation rules, implemented in a prototype, to realistic applications are presented.

CHAPTER 6

EXPERIMENTAL VALIDATION

In previous chapters, a set of transformation rules for speculative execution have been developed and proven that these rules do not change the program semantics and do not extend its deadline. Applicability refers to the possibility of using speculative execution in real-time programs, which is not a guaranteed property of the speculative execution rules as are timeliness and semantic correctness. It may happen that the preconditions of the rules are never met. Consequently, applicability cannot be validated using formal verification. In this chapter, various coding characteristics that affect the applicability of speculative execution in real-time programs are investigated using simulation. In the next chapter, prototyping is used to address applicability and usefulness of speculative execution in actual real-time applications.

We have investigated the use of speculative execution, and shown that it can be effective for an industrial real-time application, such as a cardiac workstation [98]. However, the applicability of speculative execution can potentially be affected by a number of code parameters, dependent themselves upon application domain, module type, and individual and team coding styles and practices. Since it is difficult to obtain a suitable variety of real-world programs, we have decided to investigate the effects of coding parameters through simulation. The simulation uses randomly generated sets of real-time programs created by a workload generator. We have looked at real-time programs for air traffic control [29], passive sonar [86], navigation control [31], multi-motor control system [9], and quality monitoring [28]. The design of the simulation, in large, is guided by these applications.

Among the factors that affect the applicability of speculative execution to real-time software, are data dependence, frequency of conditionals and while loops, and

the size of conditional clauses and while loop bodies. This chapter provides a study of the impact of such properties on *the number of opportunities*, *program timeliness*, as well as *performance enhancement* anticipated while applying speculative execution transformations.

In this chapter, the simulation design is discussed, and the experimental results of applying the transformation rules is presented, highlighting the impact of various parameters affecting the overall success rate of transformation. *Section* 6.1 illustrates the design of the simulation. The results are presented and analyzed in *Section* 6.2.

6.1 Design of Simulation

The experiment consists of the following steps:

1. Generate programs.
2. Assign times to statements.
3. Calculate the worst-case execution time (WCET) and the deadline.
4. Calculate the execution time without speculative execution (T_{NSE}).
5. Apply speculative (shadow) transformation rules and calculate the execution time (T_{SE} *and* T_{shadow}).
6. Compare the results of (4) and (5), and determine the effect of speculative execution on the timeliness of programs (missing or meeting deadlines).

Each step is illustrated starting with program generation through a workload generator.

6.1.1 Generating Programs

A program is a group of statements selected out of the following; IF, WHILE, ASSIGNMENT, CALL, BLOCKING_CALL, READ, and WRITE. The syntax obeys the grammar shown in *Figure* 6.1. The frequency of each type of statement is controlled by a probability density function. Based on experience with the code of real-time systems such as patient cardiac monitoring [98], air traffic control [29], passive sonar [86], navigation control [31], multi-motor control system [9], and quality monitoring [28], The following probabilities are assigned to each statement type:

IF	10%
LOOP	10%
ASSIGNMENT	35%
CALL	20%
BLOCKING_CALL	5%
READ/WRITE	20%

Both READ and WRITE use buffers. Consequently they are considered non-blocking. Calls are an invocation of code on different processor, and can be blocking (BLOCKING-CALL), or non-blocking (CALL). Parameters to calls are randomly selected from the set of variables and classified randomly as *in* or *out* parameters.

Loops and if statements are not primitive statements, in the sense that they contain more than one statement. To study the impact of the block size on the simulation results, two simulation parameters are defined to control the size of blocks within loops and conditionals by generating number of statements less or equal to these constants. Loops have an upper bound on the number of iterations which will be used in the next step to compute the worst-case execution time.

```
<program>       ::= <block>
<block>         ::= {<statements>
<statements>    ::= [<statement>]+
<statement>     ::= <if>
                  | <loop>
                  | <assignment>
                  | <read>
                  | <write>
                  | <call>
                  | <blocking-call>
<if>            ::= IF (<var>) <block> ELSE <block>
<loop>          ::= WHILE (<var>) <block>
<read>          ::= READ (<var>)
<write>         ::= WRITE (<var>)
<call>          ::= CALL ([<var>]+)
<blocking-call> ::= BLOCKING_CALL ([<var>]+)
```

Figure 6.1 The Grammar of the Generated Programs

To represent the relationship between the conditional variable and the preceding code, both loops and if statements are preceded by calls. The conditional variable is selected randomly from the *out* parameters of the preceding call. [1] For while-loops, the same call will be included in the body of the loop to update the condition variable.

Locality of reference entails some combination of lengths of live ranges and degree of reuse of variables. In the experiment, locality of reference have been treated as a measure of the first of these, and to have a separate "degree of reuse" parameter. To simulate locality of reference, a program is divided into segments. The segment size can be controlled by a constant determining the percentage of locality *(i.e., segment_size = program_size * (1-locality))*. The variables used in the program are divided throughout segments *(i.e., segment_variables = max_no_variables * (1-locality))*. A random number is used to determine how many variables from the preceding segment will be reused in the current segment. A counter is used to keep track of

[1] In principle, it could depend on a combination of parameters, but this combining is assumed to occur inside the call.

the current segment, and variable references are selected from the subset associated with that segment. For example, a program of 500 statements using 50 variables with 90% locality has *segment_size* of 50. For each segment, 5 different variables are used. Some of the variables are taken from the previous segments. Assume that the percentage of variables of reuse in each segment is 40%. Thus $5 + 5 \times 40\% = 7$ variables are used in each segment, where two of them are from the previous segment.

Two lists of variables are maintained for each statement: First, the set of variables reached (used) in that statement; second the set of all variables modified or defined. These lists will be used while applying the speculative execution rules to ensure the dependency conditions.

After generating the program control flow graph, the next step is to associate with every statement its worst-case execution time.

6.1.2 Assigning Times to Statements

At step 2, execution time is attached to each statement. For a primitive statement, the execution time is assumed to be proportional to the number of variables used or modified in that statement. Consequently, the execution time of a primitive statement can be computed by multiplying the number of variables involved by a constant. On the other hand, the execution time of a block is the sum of execution times of statements in that block.

As in real-time systems, worst-case execution time is of most interest, the next step is to analyze the generated program control flow graph to calculate an upper bound on the execution time.

6.1.3 Calculating WCET and Deadlines

The worst-case execution time (WCET) is the sum of worst-case times of all the statements of a program. For primitive statements, WCET is the assigned execution time, as discussed at the previous step. For conditional statements, the time of the longest path is used as WCET. The sum of times of the loop block statements multiplied by the upper bound of the loop index is the WCET of a loop.

Deadlines are constructed as a function of the worst-case execution time. In the experiment, after computing WCET of a program, the deadline is selected randomly in the range of $[.8, 1.1] * \text{WCET}$. Thus, a mixture of processes can be provided, some of them meeting and others missing deadlines. Selecting very tight or loose deadlines makes most programs fall into group 0 or group 1 respectively.

While timeliness of real-time systems is closely related to the worst-case execution time, speculative execution mainly contributes to improvements in the average execution time. Next the average execution time of the generated programs is calculated in order to measure the effectiveness of the proposed transformations.

6.1.4 Calculating the Average Execution Time

At this step, the (non-speculative) average execution time (T_{NSE}) is computed as follows. The average execution time is accumulated by going through each program statement by statement. For primitive statements, the execution time from step 2 is used. Care is needed for conditionals and loops. The probability of selecting the longest branch of an *if* is assumed to be 90%. For loops, a random number will be generated in the range of one to the upper bound of loop iterations. This number is considered as the average number of iterations of a loop. So the average execution time of a loop is the average number of iterations multiplied by the average execution time of the loop body per iteration.

To measure the impact of speculative execution, the average execution time of a program is compared with the execution time of the program after application of the transformation rules.

6.1.5 Applying Transformation Rules

In this step, the compiler transformation rules are applied to the generated programs. Success is monitored using two measures, applicability and performance improvement. Applicability is indicated by the number of successful applications of the safe speculative execution transformations. Performance improvement is measured by the enhancement in average execution time compared to the original program.

To obtain the average execution time using speculative execution T_{SE} (T_{shadow}), the algorithm of the previous step is used for serially executed code. However, when applying the transformation rules, parallel execution is considered as well as overheads for forking and joining. Forking and joining time, t_f and t_j, include inter-process communication, which may involve sending messages over links in case of shadow execution. If the code is running speculatively on the current processor, t_f and t_j are calculated as a linear function of the number of variables to be stored and restored (e.g., $t_f = constant_1 * sent\_variables + constant_2$). The values of such constants for running speculative code on a shadow may differ from the values for running the code on the current processor. In case of shadow execution, communication time is calculated as the sum of propagation delay, transmission delay and preparation of frames, which may vary according to network traffic. However, assuming no message contention, t_f and t_j are compile-time computable. Moreover, frames can be quickly constructed. Only a small and fixed number of variables occur in the parts of IF and LOOP constructs to be speculatively executed, and only those variables need to be involved in forking and joining. Thus most values

in a frame are known except parts of the data and the cyclic redundancy check (CRC). Assuming Ethernet connections without repeaters, the propagation delay is $500m/2 \times 10^8 m/sec$, which is $2.5\mu sec$; assuming at most 526 bytes for the variables, the transmission time is at most $526/10Mbps$, or $20\mu sec$ [47]. The preparation of frames involves local computation, and the time required is negligible. Thus both t_f and t_j are assumed to be $22.5\mu sec$.

To be consistent, the same assumptions are used as in calculating T_{NSE}. The number of loop iterations, generated from step 4, is used to calculate the average execution time of loops. Also, the longest branch of a conditional is assumed to be taken 90% of the time. To capture the number of successful matches and applications of the compiler rules, a set of counters is used to keep track of success and failure of every rule.

The final step of the experiment is to study the impact of speculative execution on the timeliness of real-time programs, as explained in the next subsection.

6.1.6 Determining Missing or Meeting Deadlines

After computing T_{SE} and T_{NSE}, programs are classified into 4 groups as follows:

Group 0: programs which miss deadlines without speculative execution and with speculative execution, i.e. both T_{SE} and T_{NSE} are greater than the deadline.

Group 1: programs which meet deadlines without speculative execution and with speculative execution, i.e. both T_{SE} and T_{NSE} are less than the deadline.

Group 2: programs which meet deadlines using speculative execution but miss deadlines with no speculative execution, i.e. T_{NSE} is greater than the deadline and T_{SE} is less than the deadline.

Group 3: programs which miss deadlines using speculative execution but meet deadlines with no speculative execution, or in which no speculative execution can be performed.

Deadline	Group 0	Group 1	Group 2	Group 3
$[.5, .7] * \text{WCET}$	1000	0	0	0
$[.7, .9] * \text{WCET}$	93	370	537	0
$[.9, 1.0] * \text{WCET}$	63	650	287	0
$[1.0, 1.1] * \text{WCET}$	0	1000	0	0

Figure 6.2 Speculative execution helps programs meeting deadlines

The programs of group 0 are of no interest since they miss deadlines for both speculative execution (SE) and non-speculative execution (NSE). No programs belong to group 3 where deadlines are missed using speculative execution but met using no speculative execution, since no code will be speculatively executed if it is not safe or profitable according to the analysis performed by the compiler. It is interesting but not surprising that every program presented at least one opportunity for speculative execution.

The selection of deadlines affects the size of each group. For instance, more programs belong to group 0 if the deadline is in the range of $[.6, .7] * \text{WCET}$ than the range of $[.8, 1.1] * \text{WCET}$. An experiment is conducted using 1000 programs, assuming 5% to 10% ifs and loops. The result of this experiment is shown in Figure 6.2.

Note that the sum of groups 0 and 2 is the number of programs missing deadlines before applying the transformation rules, while the sum of groups 1 and (the first subgroup in) 3 is the programs meeting deadlines before applying the transformation rules. For example, the third row indicates that before applying speculative execution, there are 370 (630) programs meeting (missing) deadline and (370 + 537 = 907) meeting deadlines after applying the transformation rules.

As indicated in Figure 6.2, speculative execution may not help if the deadline is $[.5, .7] * WCET$, when no programs meet their deadline. On the other hand,

speculative execution will have a minor or no effect on the timeliness of programs, if all deadlines are slack, that is, if each deadline lies in the range $[1.0, 1.1] * WCET$.

In the following section, deadlines are assumed in the range $[.8, 1.1] * WCET$ to capture the effects of deadlines on applying speculative execution.

6.2 Performance

In the simulation, the goal is to capture the impact of various parameters that affect the usefulness and applicability of speculative execution. The parameters of interest are *locality of reference for variables, program size, percentage of if statements, percentage of loop statements, the size of a branch of a conditional, the size of the loop body block, and percentage of blocking calls.* The performance is measured by speedup, improvement of timeliness and applicability, defined as follows:

- *Speedup*: the percentage of $1 - \frac{T_{SE}}{T_{NSE}}$ where T_{SE} denotes the execution time of a program with speculative execution and T_{NSE} denotes the execution time of a program without speculative execution.
- *Improvement in timeliness*: the percentage of programs which originally miss their deadlines but meeting deadlines using speculative execution.
- *Applicability*: the number of successful *while* and *if* transformations, divided by the total number of *while* and *if* statements in the considered programs.

An experiment with 1000 programs is performed. Each program contains 1500 statements, selected according to the probability density function described in *Subsection* 6.1.1. A set of 100 variables are used in each program. Deadlines are

in the range of $[.8, 1.1] * WCET$, as explained in the previous section. A locality of reference of 10% with 50% reuse of variables from the previous segment is used. [2]

To capture the impact of various parameters multiple experiments are performed changing one parameter at a time. In the following figures, SE and SH are used to denote speculative execution and shadow execution respectively. The results are summarized beginning with the effects of program size,

Program Size. Since the frequency of *if*, and *while* as well as other statements are selected according to a probability density function, the program size is not expected to have any impact on the applicability of speculative execution (percentage of success relative to the number of opportunities). The simulation results confirm our expectation, as shown in *Figure* 6.3. *Figure* 6.4 shows that speedup and improvement of timeliness stay almost the same. Since program size, in general, does not affect the rate of opportunities, both applicability and improvement scale without problems.

Size of Various Blocks. While speculative execution is possible if the size of the block to be executed is reasonably large to compensate for the overhead of forking and joining, data dependence increases and may affect the feasibility of speculative execution. In addition, chances are increased that an *if* or *while* block will contain a blocking-call which makes the satisfaction of the blocking preconditions impossible. The effect of the size of the then-clause of a conditional on applicability and other performance measures is presented in *Figure* 6.5, and 6.6. As the size of the *if* block is increased, opportunities for transformation decrease, hence the amount of speedup and improvement in timeliness is reduced. This is largely because larger blocks have higher probability of including blocking calls. Moreover, they tend to have larger variable read and write sets, which makes it more likely that these sets overlap with

[2]Every program segment refers to 10% of the variables and half of them are taken from the previous segment.

the *out* parameters of the preceding call, and also increases the time to store and restore (for shadow execution, send and receive) times. The same argument holds for effects of the size of a while-loop body, as shown in *Figure* 6.7, and 6.8.

Figure 6.9 presents the impact of the size of the else-clause of a condition on the applicability of speculative execution. The more the execution time required for the else-clause, the more expensive the rollback.

Frequency of Various Statements. Increasing the frequency of *if* statements enables more opportunities for speculative execution. However, it does not mean that on average the number of opportunities will increase. Studying the effect of changing the frequency of *if* statements, the applicability of the speculative execution transformations is found to increase (see *Figure* 6.10). However, improvement in timeliness and speedup is getting saturated and even decreases after a certain threshold, as shown in *Figure* 6.11. The reason for this phenomenon is that increasing the number of *if* statements makes nesting of conditions more frequent. On the other hand, increasing the frequency of *while* loops always has positive effects on the number of safe opportunities (see *Figure* 6.12), and on the speedup and timeliness (see *Figure* 6.13).

As expected, with the increase the percentage of blocking calls, the opportunities for transformation decrease (see *Figure* 6.14). The greater chance of having blocking calls inside *if* and *while* statements results in difficulty of satisfying the blocking constraints.

Locality of Reference. The effect of changing the degree of locality of reference and degree of reuse in the program can be seen to be non-linear. With high locality and low degree of reuse, almost all variables are local to a single segment, so few if any will "leak" into nearby segments, and there will be few data dependencies

between segments. In contrast, with low locality and high degree of reuse, almost all variables are global to the program as a whole, but variables are referenced essentially at random, so with a large variable set, there will again be little overlap between segments.

Overall, as locality decreases, opportunities for speculative or shadow execution increase for short-lived constructs, and decrease for long-lived constructs (see *Figures* 6.15 and 6.16). An overall increase is expected, since the opportunities studied in this simulation are mostly in short-lived constructs.

As a general observation, shadow execution always outperforms speculative execution, because rollback is not necessary for shadow execution, which makes the satisfaction of its timing precondition easier. In addition, the shadow processor is under-utilized, which suggests that a shadow processor can serve more than one process.

Through the simulation, it has been shown that speculative execution can enhance the timeliness and performance of real-time programs. Data dependence has a significant impact on the applicability of speculative execution. Opportunities of speculative execution scales with real-time program sizes. The higher the frequency of blocking calls, the smaller the number of opportunities. Applicability of speculative execution tends to diminish for large sizes of conditional and while loop body blocks. As expected, shadow execution is always more applicable and profitable since rollback is not required. In the following chapter, a prototype implementation of the transformation rules is described and a study of the applicability and usefulness of speculative execution in actual real-time applications is presented.

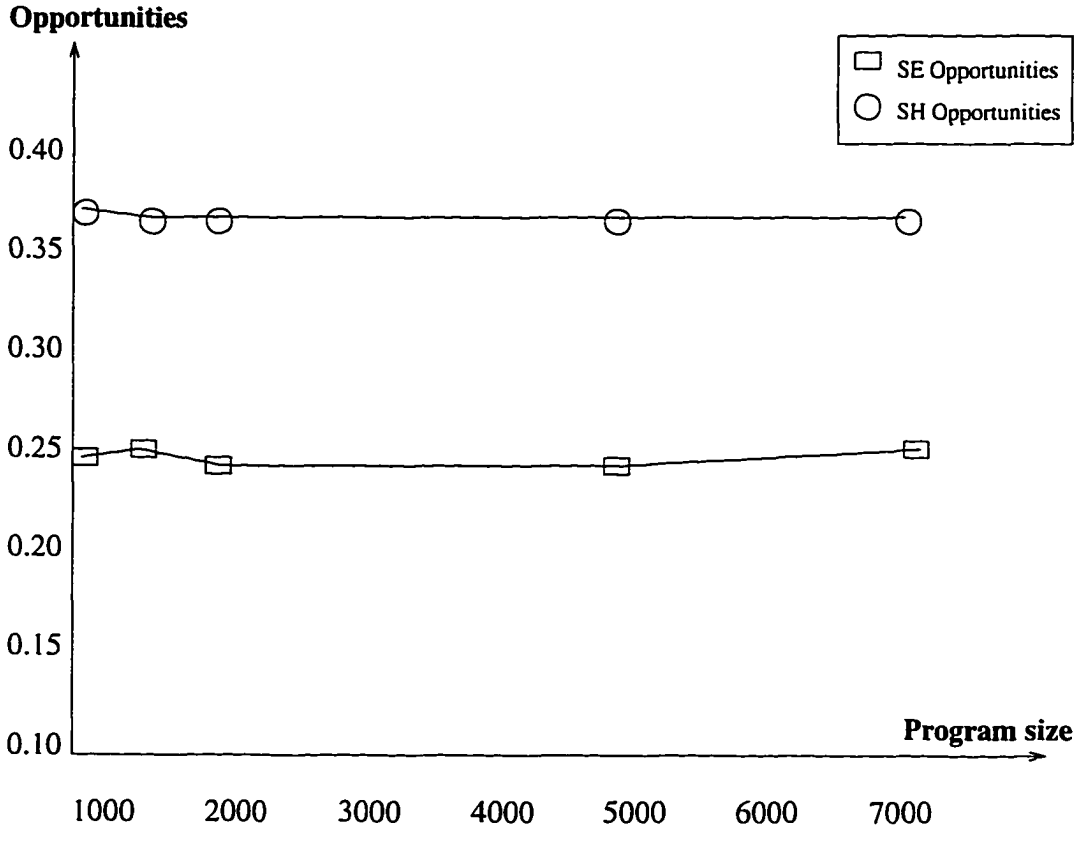

Figure 6.3 The relationship between opportunities and program size

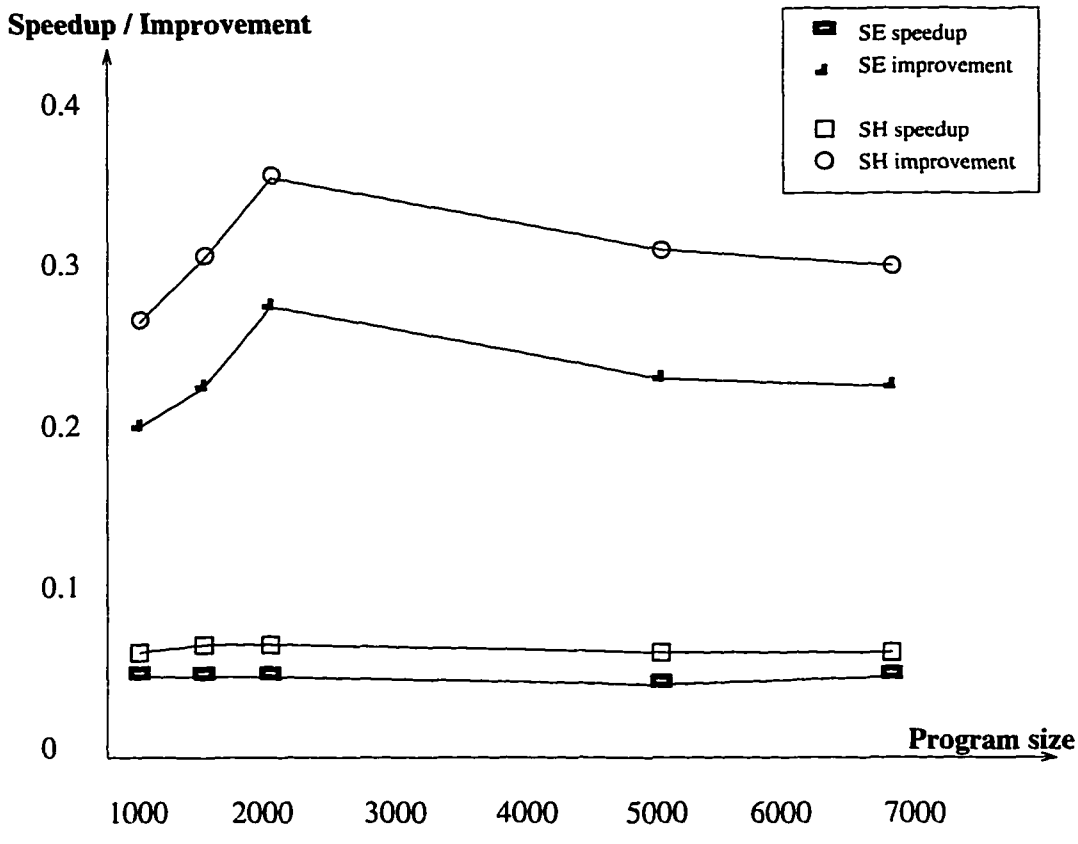

Figure 6.4 The effect of program size on performance

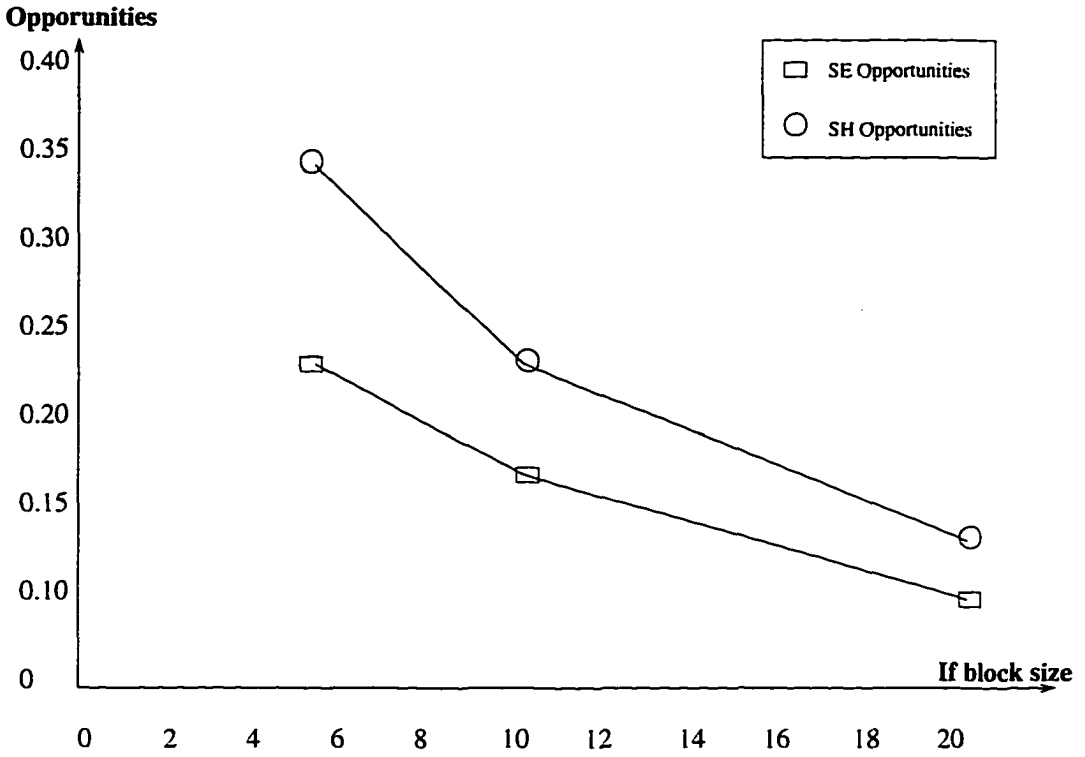

Figure 6.5 Size of *if* blocks versus opportunities

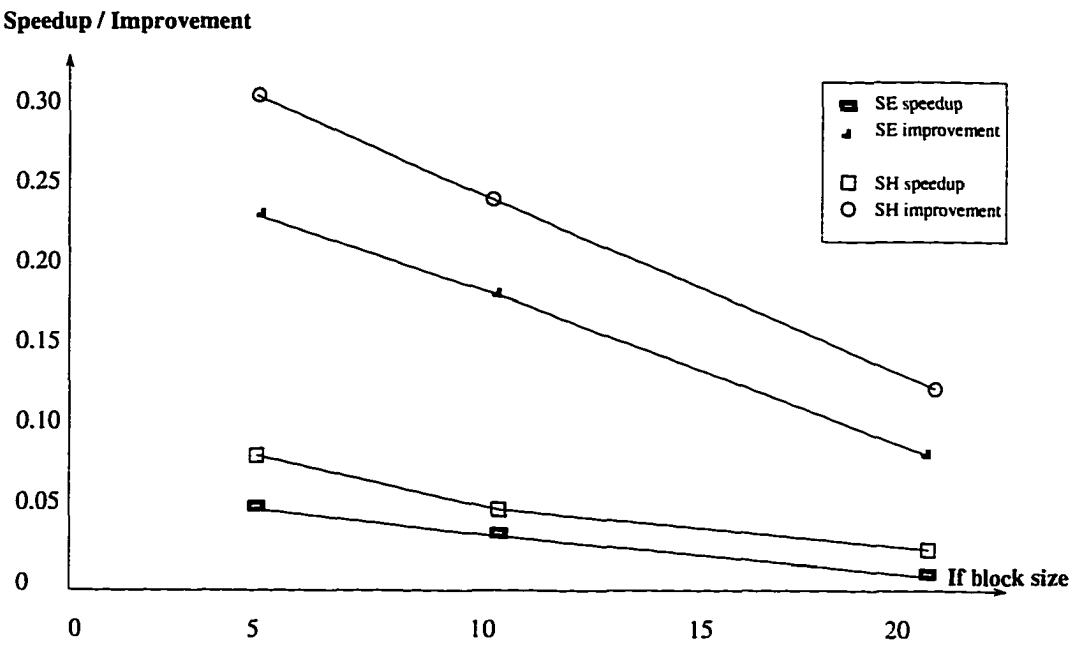

Figure 6.6 Impacts of the *if* block size on performance

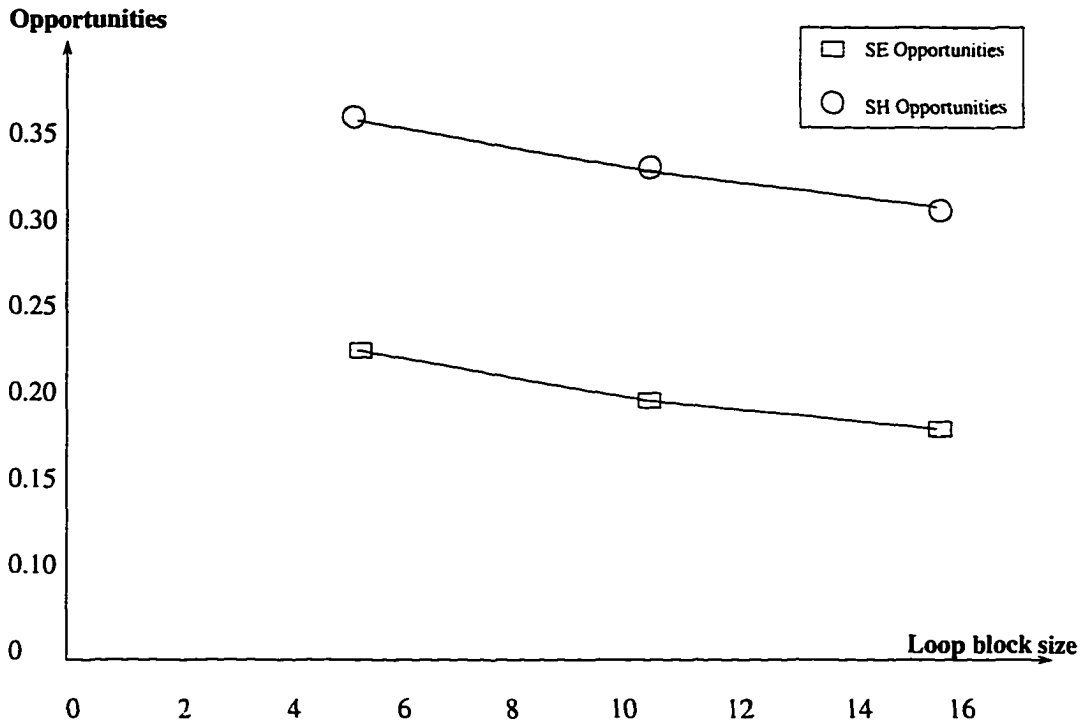

Figure 6.7 Size of *while* blocks versus opportunities

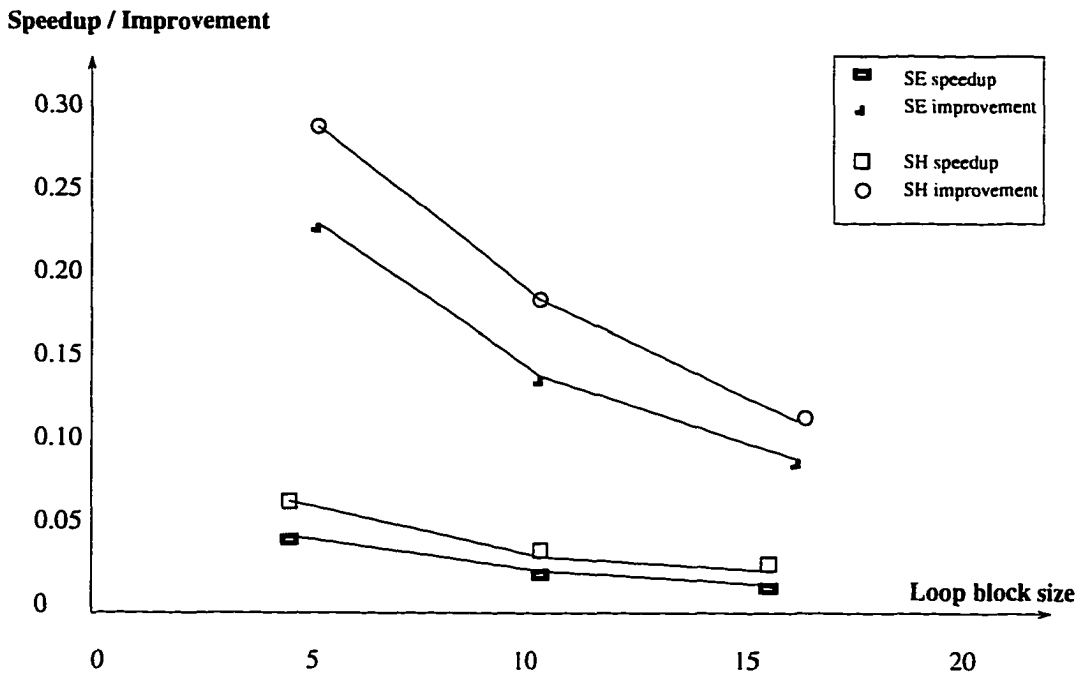

Figure 6.8 Impacts of the *while* block size on performance

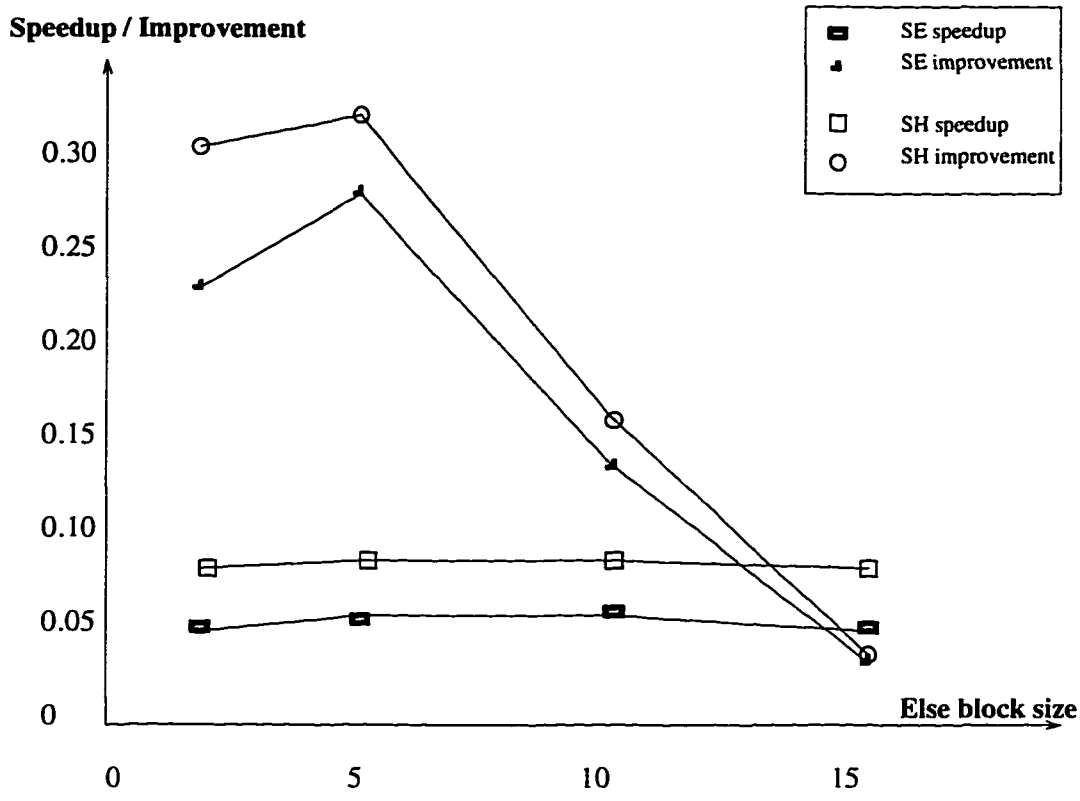

Figure 6.9 Size of else block versus opportunities

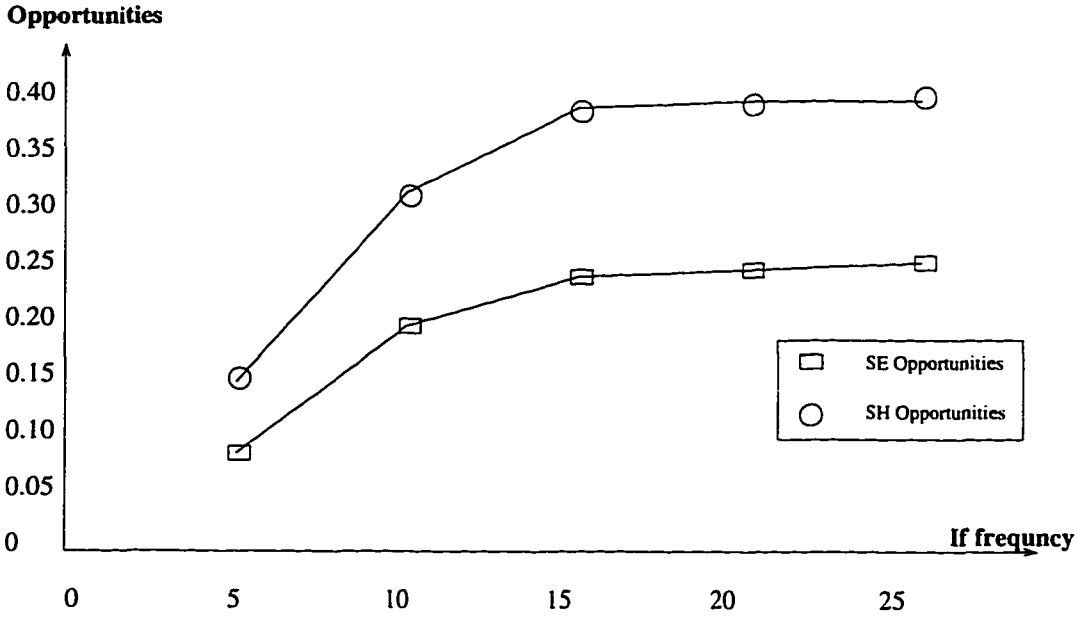

Figure 6.10 *If* frequency versus opportunities

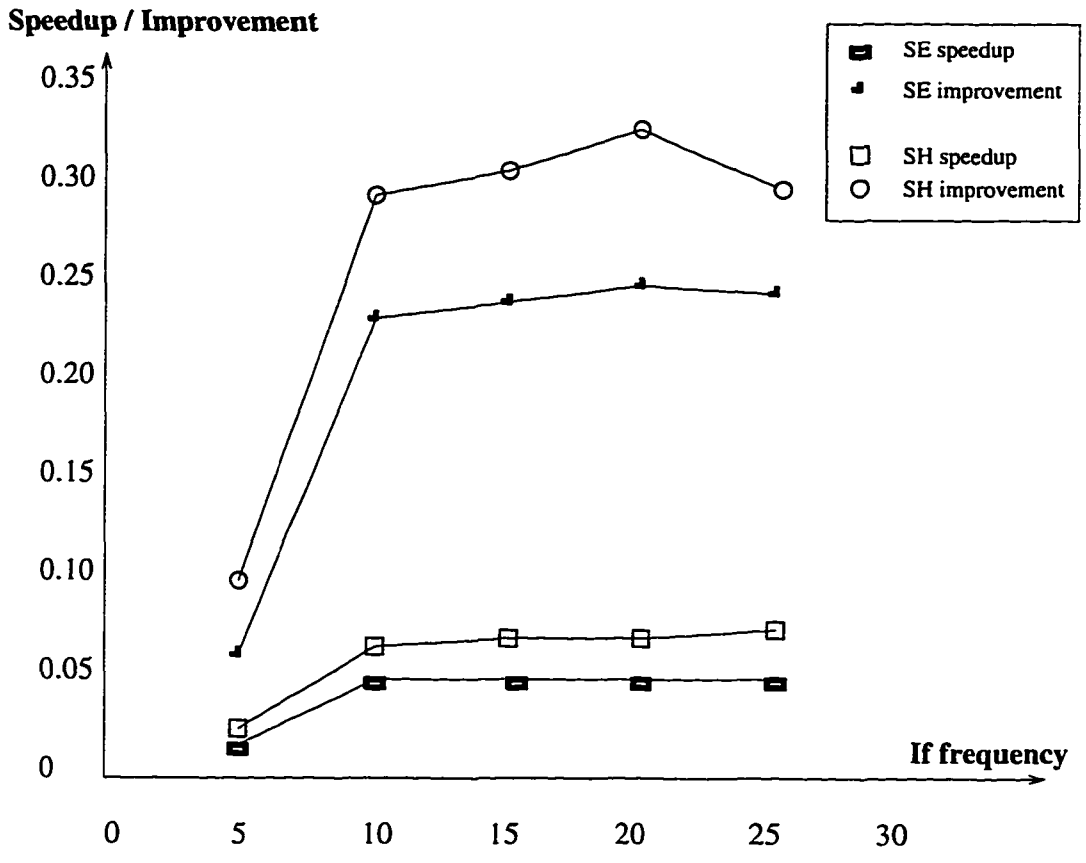

Figure 6.11 Impacts of *if* frequency on performance

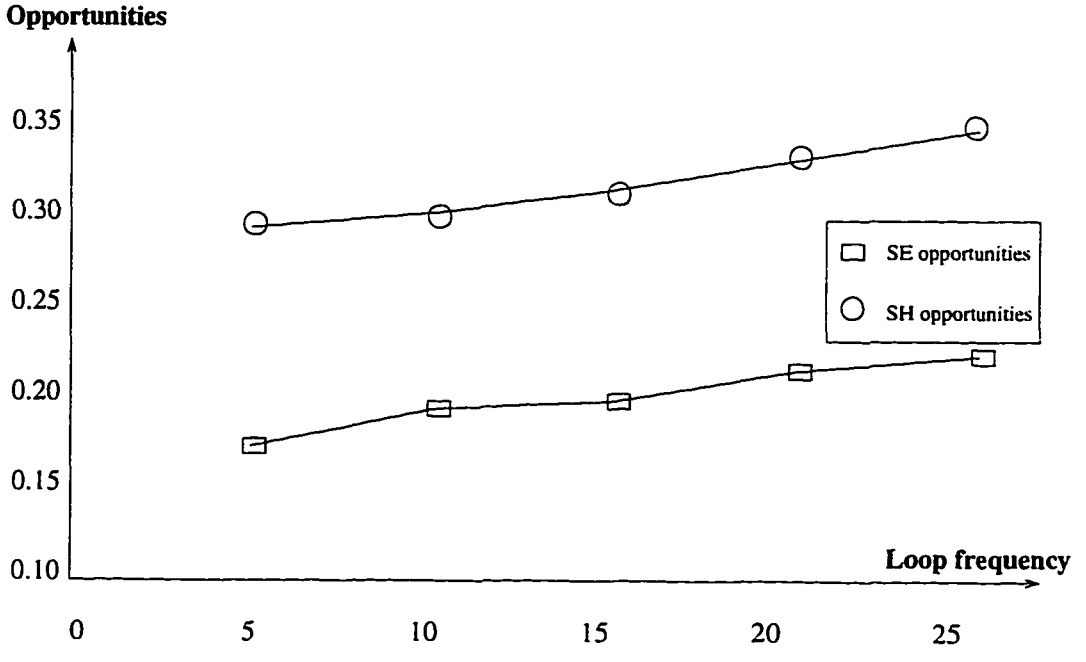

Figure 6.12 *While* frequency versus opportunities

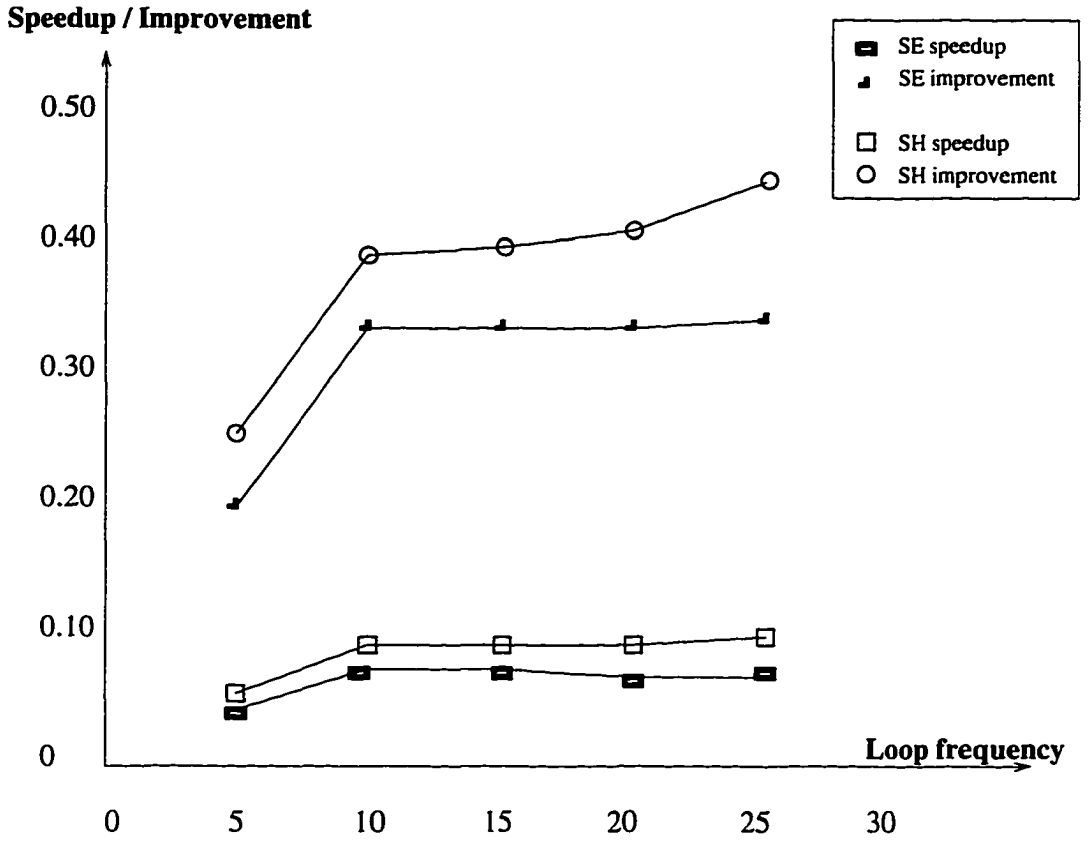

Figure 6.13 Effects of *while* frequency on performance

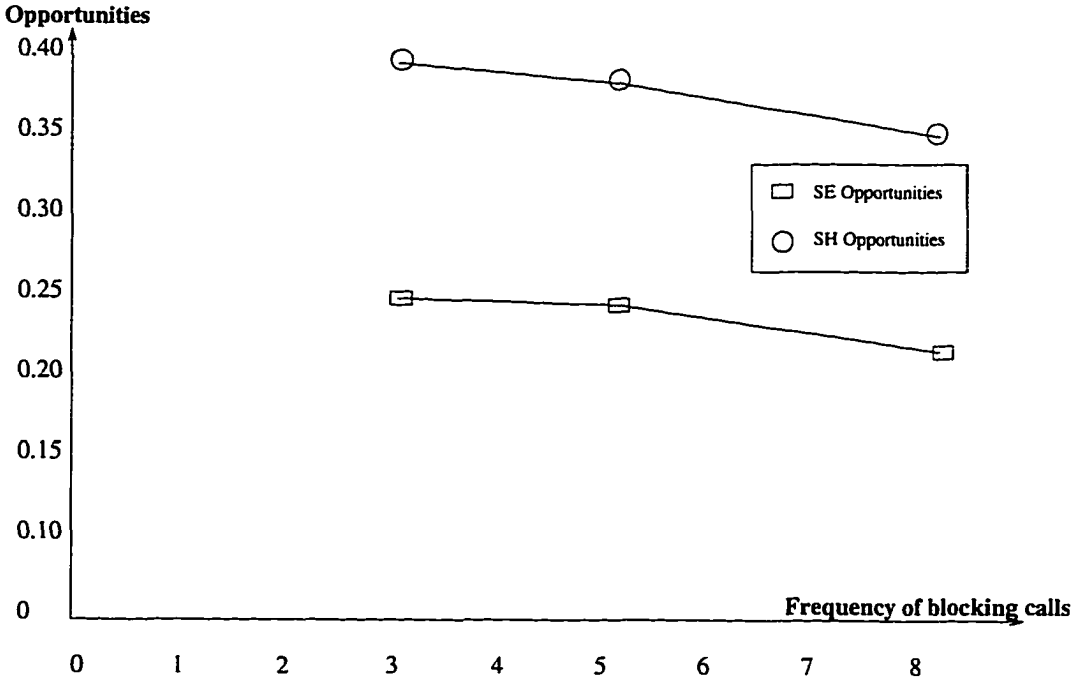

Figure 6.14 Percentages of blocking calls versus opportunities

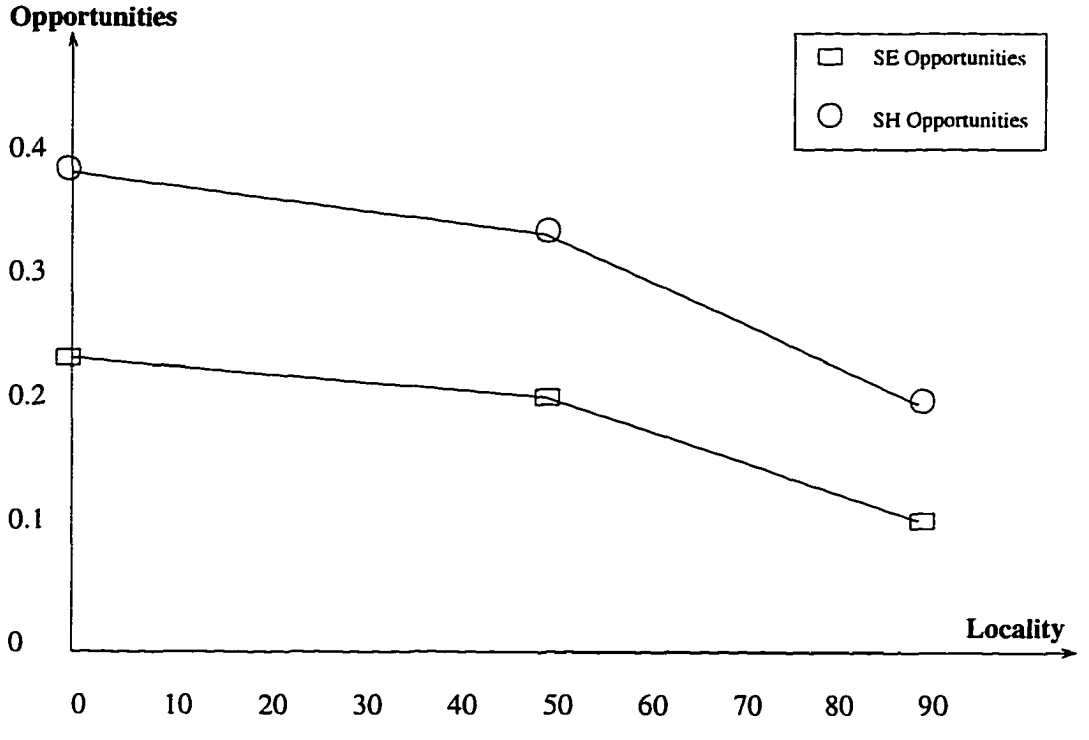

Figure 6.15 Locality of variables reference versus opportunities

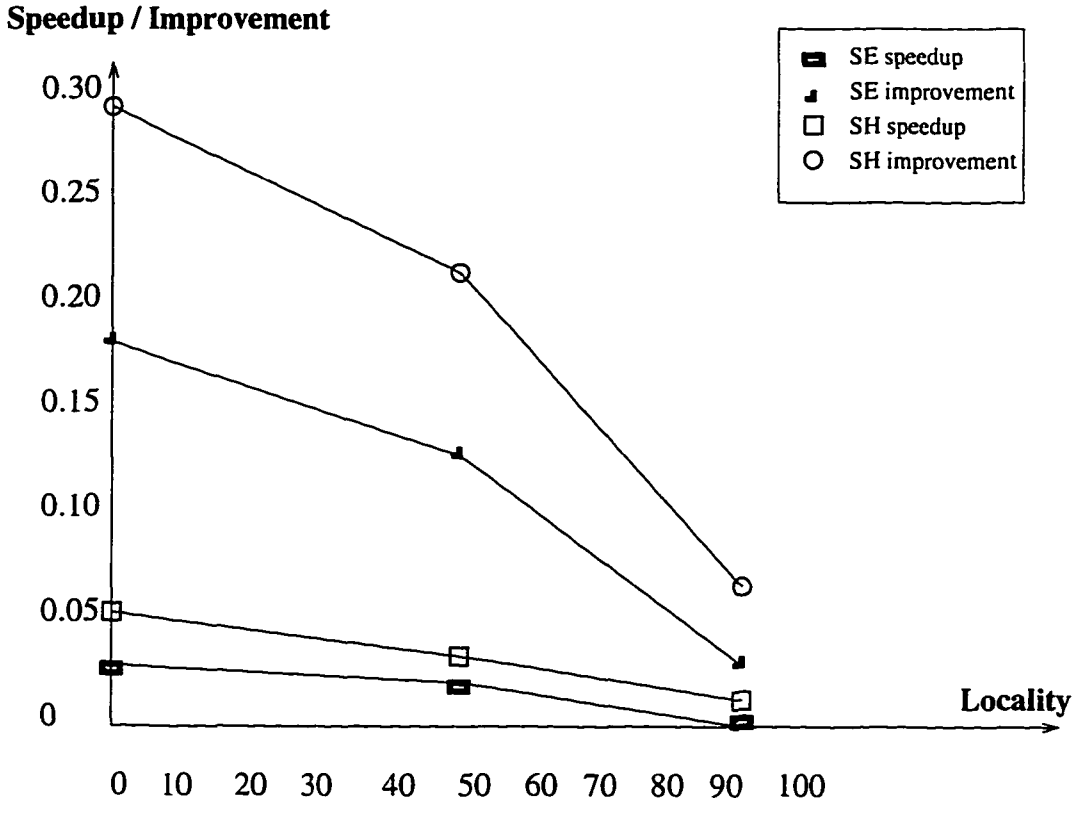

Figure 6.16 Effects of locality of variables reference on performance

CHAPTER 7

IMPLEMENTATION AND TEST ENVIRONMENT

In the previous chapter, the impacts of various code characteristics on the applicability of speculative execution have been studied. While the simulation results clearly show the potential and usefulness of speculative execution in real-time systems, we decided to go for the more aggressive validation by applying the transformation rules to actual real-time applications. This chapter includes a description of the prototyping efforts to implement and test the applicability and usefulness of the transformation rules of *Chapter* 4 in fragments of actual real-time applications.

As illustrated earlier, the speculative execution transformations are based on a language model which supports only predictable constructs and expresses timing constraints on fine-grained level. A prototype based on a new object-oriented language for complex real-time applications, called CRL [99], is being built at the Real-Time Computing Laboratory at NJIT. The prototype includes a compiler, a transformation engine supporting various types of transformations such as conditional linking and partial evaluation and others discussed in *Chapter* 3, an objects-to-processors assignment tool, and a run-time environment. The execution of processes is handled by a single kernel. The kernel manages object queues, and initiates execution of various methods (calls). The processor interconnection topology and network is simulated by a separate tool communicating with the kernel to get requests and to simulate message propagation delays and queuing.

This chapter is organized as follows. First, an overview of the components of the experimentation platform is provided. Next, the language characteristics are addressed, emphasizing why they support the features assumed in the language model of this work. Then, the compilation process is described for programs written

in that language. The timing tool supporting the analysis is discussed, followed by a description of the compiler transformation engine. Then, the role of the schedulability analyzer is described. The discussion of the run-time environment begins by presenting the design of the kernel, followed by the network architecture simulation and user interface. Finally, the implementation of the compiler transformation rules for speculative execution is illustrated, followed by a summary of experimental results.

7.1 Overview of the Platform Components

The platform consists of seven major components, as shown in *Figure* 7.1. In this section, each component is briefly defined. A more detailed discussion follows in the balance of the chapter.

The input for the prototype may include, in addition to source code, an architecture file describing the target processors' architecture, an instruction time map for that architecture, and an assertions file providing user annotations to be used by the transformation engine (for example, in performing partial evaluation).

The first component is the *compiler* for the CRL language. The front end of the compiler generates intermediate code (in this implementation, a safe subset of C++), including run-time checks, and creates files containing constraints and assertions, and some additional information, for the run-time environment. The compiler also generates a representation for the call graph (caller and callee relationships), a data dependence graph, and control flow graphs of processes to be used both by the timing tool and by the transformation engine. Currently no machine code is generated, relying on the intermediate code in the analysis. The timing tool then uses the instruction timing map to assign times to atomic statements (but not structured statements or calls) of the intermediate code. The timing tool annotates every

statement of the intermediate code with its execution time, implemented by defining a time variable incremented past statements of every basic block by the execution time of statements of that block, and output a timed intermediate code.

The *analysis/transformation* engine uses the timed intermediate code generated by the timing tool and applies static analysis and various transformations, as discussed in earlier chapters, to improve the code. Moreover, it tries to eliminate some checks, and to detect certain classes of errors, resulting in a final version of the code and of the constraint file.

The *schedulability analyzer* then takes the transformed code and constraint file, and certifies schedulability under the validity of constraints and assertions. The schedulability analyzer also reports a possible object-to-processor assignment (in which some objects may be cloned, or even replicated on every processor), and a partial or complete static scheduler.

The *run-time preprocessor* (linker) translates the intermediate code into executable code. The *run-time kernel* uses the executable code and the final constraint file and consults the static schedule generated by the schedulability analyzer to schedule tasks, allocate resources, and manage object queues. The *network simulator* provides the kernel with the delays due to communications (transmissions and message queuing). Finally, the *user interface* component displays some measurements, such as performance, processes missing deadlines, and average case improvement.

The implementation efforts related to this thesis have been focused on the analysis and transformation engine. The interaction with the assignment tool is a future extension. Currently, the run-time component is considered as a test environment, used to report measurements of the applicability and profitability

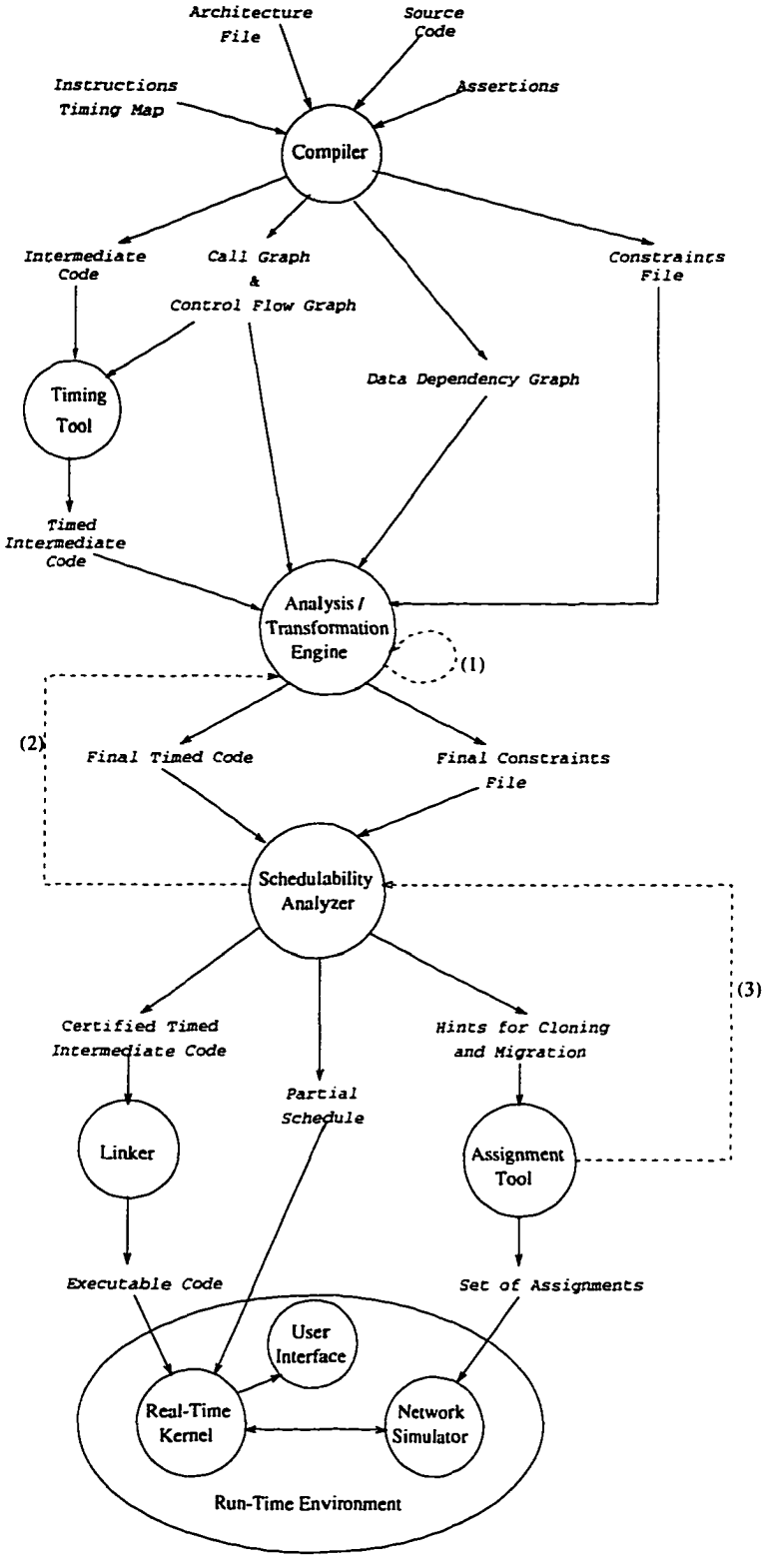

Figure 7.1 The platform software components

of various techniques involved. In the future, the run-time environment may be upgraded to reflect additional issues, as for example scheduling disciplines.

Before illustrating the compiler as the first component in the prototype, an overview of the CRL language, on which the platform is based, is provided.

7.2 The Real-Time Language

As just mentioned, the experimentation prototype is based on a new real-time language, CRL. The language is sufficiently robust and expressive to capture most standard functionality, but also sufficiently structured in both native constructs and annotations to afford static analysis by reasonable techniques. The language serves as a vehicle for research and experimentation in real-time languages, schedulability analysis, techniques for enabling efficient analysis, and assignment of objects and processes to processors for complex real-time programs.

In this section, an overview of the language is provided as well as its real-time features. The language expressivity in stating a wide range of timing constraints is elaborated, with an argument of why it accords with the language model of this thesis. Finally, an example of a program written in that language is provided.[1]

7.2.1 Language Overview

A run-time program in CRL can be seen as a collection of strongly-typed objects, instantiated from abstract data types (classes). An object encapsulates a persistent state, exported operations on this state, internal operations, and possible threads of control. A thread may call operations of the owner object and exported operations of other objects. There are primitives for synchronizing persistent states of objects: critical sections for threads of the object and caller queues for external threads.

[1]The discussion in this section is largely based on [99].

For the purpose of our current research, basic types *integer, rational, boolean,* and *character* suffice. There are compound type constructors *array,* and *record.* Arrays have syntactically declared rank, and compile-time specified bounds. There are no pointers, and in particular, no dynamic structures, and no *new* or *free* operators for record types. Array index expressions currently may involve only scalar variables.

The expression and operator grammar is standard. Timing for an expression involving only register or immediate operands is deterministic, and depends only on the operator and the type and storage class of the operands.

The control flow constructs CRL provides includes *loop, exit, if-elseif-...-else.* It is required that all loops and recursion are a priori bounded. These bounds may be specified: (i) explicitly; (ii) by user assertions (in which case it is a run-time error if such an assertion fails); (iii) at compile-time through static analysis; or (iv) at or before link-time by partial evaluation. Non-complying programs are not compiled.

Parameters may be passed *IN, OUT, INOUT.* IN parameters are passed by value; OUT parameters are passed by result; INOUT parameters are passed by reference rather than value-result. Restrictions in the type system ensure that alias analysis is reasonably precise, while avoiding the semantic checks required for use of value-result parameters; the only possibility for aliasing is through method reference parameter collisions.

The syntax uses single-line prefixed comments (rather than delimited comments); the comment character is %. Any input between a comment character and the next end-of-line is ignored by the compiler.

In this section, the structural constructs of the the CRL language is discussed. The following section shows how timing constraints are expressed.

7.2.2 Observably Timed Statements

The set of program statements whose execution time may occur in timing constraints is specified syntactically. All critical section accesses, accesses to I/O, and synchronizations/messages between processes are by default observably timable, as are the beginning and the end of each process. The fork and join nodes governing one or more conditionally executable timable nodes are themselves observably timable. Other statements may be labeled as *timable*.

The execution time of a timable statements can be distinguished by labels. A timable statement has the form

< timed_statement > ::= [$ < label >!] < untimed_statement > [! < label > $]

where the first label represents the execution initiation time for untimed_statement, and the second label represents the execution completion time.

Constraints are either *absolute*, relating a statement in a process or object to the beginning or end of the current frame, or *relative*, constraining the time interval between two observably timable and statically co-executable statements, each in or called by the same process, or involved in a single inter-process synchronization. Constraints are either *max_time* or *min_time* constraints, accordingly placing an upper or a lower bound on absolute or relative times; more complicated constraints are constructed as combinations of these basic types.

The following section includes a discussion of the language static semantics that will be checked by the compiler.

7.2.3 Static Semantics

The language enforces standard Pascal-like static restrictions on type consistency, function arity and name conflict. A few other similar constraints on the use of names (such as the second label in time expressions) are given above. The compiler also

inserts checks for array bounds (although some may be removed by optimization) and other similar access validity enforcement mechanisms.

There are two additional semantic restrictions. First, there is a restriction on the reference of the first label on a time expression, relative to the location of the current statement. This constraint can be verified by standard attribute grammar techniques. Second, and inherent in the real-time nature of the language, constraints and assertions are subject to compile-time verification or run-time checking. In general, timing constraints must be verified at compile-time.

In addition to the timing constraints, the language allows restrictions to be imposed on the activation of processes, the number of iterations of a loop, and the depth of recursion. Process activation/deactivation constraints are enforced by the scheduler and run-time system. Iteration and recursion constraints must either be verified at compile-time or checked at run-time.

Assertions are assumed true at compile (or link) time, and may be used by the partial evaluator and subsequent analyses and transformations. It is however required that they be checked at run-time (unless proven to be redundant and removed by the compiler). It is a fatal error for a run-time check of a constraint or assertion to fail.

After giving an overview of the language, and illustrating its expressiveness for timing constraints, it should be matched with the thesis assumptions about the real time language. In the next section, real-time language model of this thesis is related to the features provided by CRL, concluding with the suitability of using it to validate this work.

7.2.4 Relating to Our Language Model

Fundamentally, both a high-level general-purpose language, and a powerful expression mechanism for timing and other constraints are needed. The CRL language provides the standard core of a high-level language, including array and record type constructors and function calls allowing recursion with an upper bound on the number of recursive calls. All language constructs are predictable in terms of execution time, either through compile-time bounds or through run-time assertions. The start and end of any atomic statement can be used as temporal reference points, allowing an extremely powerful language for temporal constraints.

Moreover, the CRL language allows concurrency and inter-process synchronization. In CRL, a real-time system consists of a set of top-level objects (some of which with threads of control), possibly running on a distributed network, accessing a set of resources managed as other objects, and synchronizing via calls and messages (messages are not yet implemented). The objects are declared on the basis of abstract data types (classes).

In addition to the above, the language provide a very strong static semantics that allows an intensive compile-time checking of all interfaces as well as for timing constraints.

The next section provides an example written in CRL.

7.2.5 An Example

To present the syntax as well as the power of CRL in expressing timing constraints, an example of aircraft navigation control system, similar to the one discussed in [31], is shown. The route of an aircraft is represented by a set of goal coordinates (stored in the GOAL array). This set of coordinates is assumed to be provided by another module, and passed as a parameter to the navigation control thread. The

algorithm can be summarized in three steps. First it samples the aircraft's current coordinates, direction (heading), roll, and ground speed [2]. Second, it consults the GOAL array for the next coordinate to target and calculates, the relative attitute and the new direction angle. Finally, it adjusts the throttle and roll to move to the new coordinate. For simplicity, a 2-dimensional abstraction of navigation control problem is considered. Assume the following timing constraints imposed by the problem:

1. Control update should be done every 20 ms.
2. All measurements updates should be done within the first 5 ms in each period.
3. All throttle and flap changes must be made within 3.1 ms of the actual ground speed reading.

The CRL code for that example is shown below. One important observation in the program is the use of labels to express the timing constraints relative to some other point in the program. The label *read_stat* is used to reference the timing constraints imposed on the execution of the block in the thread *control* relative to an earlier execution point. Another observation is the flexibility of expressing timing constraints on statement or a group of statements (block) in addition to methods and threads.

```
types

record
   vars
     rational x,
     rational y,
     boolean passed
  endvars
endrecord POINT,

array 1..100 of POINT endarray GOAL,
```

[2]While other readings may be required, for simplicity, these are only considered.

```
% Definition of the velocity class
classimplementation velocity

  export velocity
         methodinterface get
   in rational tmp
   out rational speed
         endmethodinterface get

% Method to update the current status by reading various measures
method get

    % Method interface specification
    in rational tmp
    out rational speed

    % The body should be here
    IOread(SPDMTR speed)

endmethod get

endclassimplementation velocity,

classimplementation navigation

   % Class interface specification
   export threadinterface control
              in GOAL goal
          endthreadinterface control
   import velocity

   % Constants declaration section
   consts
     100   NCOORD,
     400   VHIGH,
     0.0001 EPS
   endconsts

   % Variables declaration section
   vars
      rational x,              % Current x-coordinate
      rational y,              % Current y-coordinate
      rational theta,          % Current direction angle
      rational speed,          % Current velocity
      rational roll,           % Current roll
      rational throttle,       % The aircraft throttle
```

```
      velocity vel              % An object to monitor the velocity
   endvars

% Method to update the current status by reading various measures
method update_status

   timeconstraint nolaterthan (5)       % The method has a deadline of 5
   endtimeconstraint

   IOread(GPS x)               % Read the current coordinates
   IOread(GPS y)
   IOread(NAV theta)           % Read the current angle
   call vel.get(theta,speed)         % Read the current speed through
                               % the object velocity

endmethod update_status

% Method to calculate the relative attitude and the new angle adjustment
method compRelAtt

    % Method interface specification
    in rational theta,
       rational x,
       rational y,
       rational gx,
       rational gy
    out rational rtheta

    % The body should be here

endmethod compRelAtt

% Method to calculate delta theta (angle deviation) if the velocity of the
% aircraft reaches the maximum
method safeDtheta

    % Method interface specification
    in rational rtheta,
       rational roll
    out rational dtheta

    % The body should be here

endmethod safeDtheta

% Method to compute the new flap of the aircraft based on the current roll,
% velocity and the required angle deviation
```

```
method compFlapw

    % Method interface specification
    in rational roll,
       rational speed,
       rational dtheta
    out rational wflap

    % The body should be here

endmethod compFlapw

% Method to compute the new throttle of the aircraft based on the current roll,
% velocity and the required angle deviation
method compThrottle

    % Method interface specification
    in rational roll,
       rational speed,
       rational dtheta
    out rational throttle

    % The body should be here

endmethod compThrottle

% Method to do the action of the contol
method action

   % Interface specification
   in GOAL goal
   inout integer index

   % Declaration section
   vars
      rational gx,
      rational gy,
      rational rtheta,
      rational dtheta,
      rational abs_rtheta,
      rational wflap
   endvars

   call self.update_status() !read_stat$       % read the current measurements

   block
     % Get the next target coordinates
```

```
      if goal(index).passed
       then
         gx := goal(index).x
         gy := goal(index).y
         index := index + 1 - ((index + 1) / NCOORD) * NCOORD
      endif

      % Using relative attitude w.r.t target compute angular adjustment
      call self.compRelAtt(theta,x,y,gx,gy,rtheta)
      call abs_rtheta.abs(rtheta)
      if  abs_rtheta < EPS
       then
         dtheta := 0
       elseif speed < VHIGH
            then
              dtheta := rtheta
            else
              call self.safeDtheta(rtheta,roll,dtheta)
      endif

      % Adjust flap and throttle for heading
      call self.compFlapw(roll,speed,dtheta,wflap)
      call self.compThrottle(roll,speed,dtheta,throttle)
      IOwrite(THROT throttle)
      IOwrite(FLAP wflap)
   endblock
   timeconstraint
       nolaterthanrelative (read_stat 3 local)
       % to generate the output fast enough
   endtimeconstraint

endmethod action

% The periodic navigation control
thread control

   % Specification for the period and any activation constraints
   activationdeactivationconstraint
       periodic use (20)
       firstactive nosoonerthan (5)
   endactivationdeactivationconstraint

   % Declaration section
   vars
      GOAL goal,
      rational x1,
      rational y1,
```

```
      integer index
   endvars

   index := 1
   loop nomorethaniterations 5
      IOread(user x1)
      IOread(user y1)
      goal(index).x := x1
      goal(index).y := y1
      goal(index).passed := false
      call self.action(goal,index)
      index := index + 1
   endloop
   call self.action(goal,index)

endthread control

endclassimplementation navigation

endtypes

vars
   navigation nav
endvars
```

In the following section, the input and output of the compiler are discussed. The interface between the compiler and other modules in the prototype is described. Finally, some of the restrictions assumed in order to facilitate the compilation are illustrated.

7.3 The Compilation Process

Inputs to the compilation process include (1) the source code, (2) a file of architectural specifications (for now, a homogeneous network with an arbitrary topology is assumed), including instruction-class/time maps, network topology, and other interconnection details, and (3) a (possibly empty) file of compile-time assertions for the partial evaluator. The output from the compiler will be an intermediate code program (in C++) and a timing constraints file. In addition, the compiler will

construct a *call graph*, a *data dependence representation* used by the timing tool, and the *control flow graph* to be used by the analysis/transformation engine. Currently, the compiler generates a use-def chain [2] as a data dependence representation, with monolithic handling of arrays (i.e., reference to one entry of an array is considered as using the whole array). The intermediate code is then subject to transformations by the analysis/transformation engine after being analyzed by the timing tool. Correspondence between the generated code and the control graph is maintained by two pointers per basic block to the starting and ending line numbers of the translation of that block.

Some restrictions have been imposed to facilitate the compilation process. As in Pascal, use or reference to any variable or object should be preceded by an explicit declaration of that variable or object. All parameters of objects, methods and threads should be explicitly specified as either imported or exported. The compiler will match any call to a method or a thread against the interface of that method or thread. The language provides only static scoping and at present disallows aliasing.

Currently, no target architecture for the compilation process is assumed. The transformed C++ code will be further compiled and linked with other library routines. The kernel will be responsible for invoking the generated executable code.

In this section, the compilation process as well as the interaction with other component in the prototype have been discussed. The following two sections provide a description of the tools that use the output from the compiler, namely the timing tool and the analysis and transformation tool.

7.4 The Timing Tool

The timing tool is used to provide a safe static estimate of the execution time of programs. Inputs to the tool include the timing map of instructions executed by

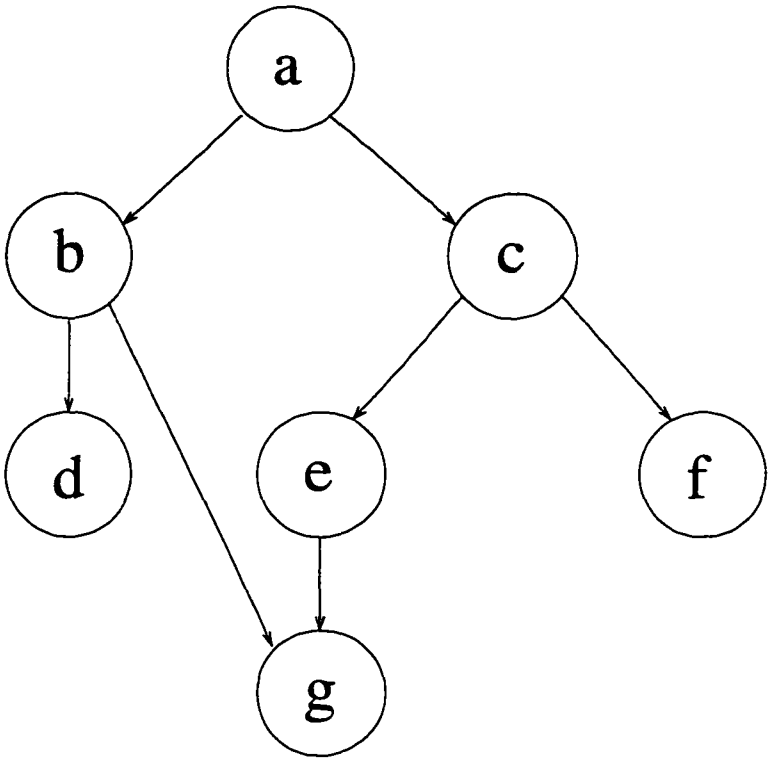

Figure 7.2 An example of a call graph

the target architecture, given as a table of instruction type and required execution time. In a heterogeneous network the instruction types of all the processor types are specified.

To resolve the execution time of calls, the timing tool uses the call graph, unwound if necessary in the presence of (bounded) recursion. It starts by computing the execution time of leaf methods in the call graph, using this information in their callers, and so on. For example, in the call graph in *Figure* 7.2, the timing tool will start with g then d,e, and f followed by b and c. Finally, it calculates the execution time for a. For remote calls, the tool should consider communication delays that messages may anticipate due to contention. An upper bound on the propagation of messages throughout the network is assumed.

The timing tool will calculate two types of execution times: first the worst-case execution time of processes, to resolve references to other methods through

calls; second, a timing annotation on every executable statement, both simple and structured, by consulting the timing map provided as an input to the platform. Currently, execution time of every basic block is stored in the control graph.

The timing tool analyzes the control graph for every method/thread to calculate the execution time of basic blocks to be used in justifying safety and profitability of transformations. In addition, the timing tool will add a statement to the output intermediate code past those corresponding to a basic block to increment a local time variable used to propagate static timing prediction to the run-time environment (see *Section* 7.8). The worst-case execution time of the whole method/thread will be deducted using execution time of basic blocks and will be stored with the method/thread entry in the call graph.

Because only intermediate code is generated in the current implementation, a map for the basic data types (classes), defined by the language, is used. The execution time of instructions (methods in the basic classes) in the map will be based on some reasonable assumptions. Compound statements and calls are annotated by the compiler with their initialization time as well as other constant execution time not including the cost of the statements in the body. For example, for a loop the compiler will annotate the initialization time which will be added to the execution time for evaluating the loop condition and a jump multiplied by the worst-case number of iterations. Assuming a network of homogeneous processors, an instruction-dependent timing map is provided.

The output of the timing tool is timed intermediate code. *Figure* 7.3 shows the timed intermediate code for the method *action* in the CRL example of *Subsection* 7.2.5. Note the increment of the time variable past every basic block. Time elapsed to evaluate conditions is reported earlier to facilitate the implementation. The transformation engine then uses that output and the timing constraints

```
void action(GOAL goal , Integer& index ) {

  long time=0;

  this->update_status();
  time += 14; // time for the call
  time += 2; // time for the condition evaluation
  if( goal[index].passed ) {
    action_gx = goal[index].x;
    action_gy = goal[index].y;
    index=index + 1 - ((index + 1) / NCOORD) * NCOORD;
    time += 10;
  }
  self->compRelAtt(theta,x,y,action_gx,action_gy,action_rtheta);
  time += 13;
  action_abs_rtheta.abs(action_rtheta);
  time += 3;
  time += 3; // time for the condition evaluation
  if( action_abs_rtheta < EPS ) {
    action_dtheta = 0;
    time += 2;
  }
  else {
    time += 2; // time for the condition evaluation
    if ( speed < VHIGH ) {
      action_dtheta = action_rtheta;
      time += 2;
    }
    else {
      this->safeDtheta(action_rtheta,roll,action_dtheta);
      time += 5;
    }
  }
  this->compFlapw(roll,speed,action_dtheta,action_wflap);
  time += 7;
  this->compThrottle(roll,speed,action_dtheta,throttle);
  time += 5;
  cout << "Write to THROT :" << throttle << endl;
  time += 3;
  cout << "Write to FLAP :" << action_wflap << endl;
  time += 3;
}
```

Figure 7.3 An example of the timed intermediate code

file generated by the compiler to check the feasibility, safety and profitability of transformations, as elaborated in the next section.

7.5 The Analysis and Transformation Engine

The transformation engine uses the data dependence graph, the call graph, and the control flow graph generated by the compiler to detect various possible code transformations. The timing constraints file is consulted to test the safety of transformations; the timing profile generated by the timing tool is used to measure their profitability.

We have implemented the transformation rules provided in the *Chapter* 4, as illustrated in *Section* 7.9. In the future, we plan to implement other transformations, such as those discussed in the *Chapter* 3, possibly including partial evaluation [72], branch/clause transformations [97], and conditional linking [98] to test their interaction with our approach. The engine will be applying the transformations as a sequence of steps. In each step a different kind of transformation will be considered. The order in which the transformations will be applied remains an issue that our future experiments will address. It may be necessary to repeat a step because of successful transformations in other steps. For example, branch/clause transformations may need to be re-applied if a condition can be eliminated by the conditional linker. This dependence is represented by the feedback arrow (1) in *Figure* 7.1.

The analysis/transformation component will have two effects: first, it will change the code according to the rules of the transformations applied; second, it may relax some of the constraints or strengthen some of the assertions. Actually, some transformations change only the final form of the code, without affecting timing constraints, like branch/clause transformations. On the other hand, others may affect both the code and the constraints, such as compiler optimization, as for example by removing unreachable code.

The output from this tool is updated timed intermediate code, as well as an updated timing constraints file. These outputs are then used by the schedulability analyzer, illustrated in the following section. However, the schedulability analyzer may need to call the transformation engine again if it is not able to guarantee schedulability.

7.6 The Schedulability Analyzer

The transformed code and the revised constraint file produced by the analysis and transformation engine are passed to a schedulability analyzer. The schedulability analyzer may use either an exhaustive or a heuristic analysis to produce an the assignment and a certificate of schedulability. The analyzer may also report a partial static schedule to be used by the run-time environment. In addition, it may generate directives for migration and cloning to the assignment tool.

The schedulability analyzer may also consult the assignment tool for the feasibility and profitability of certain transformations, as in the case of parallelization and speculative execution (feedback (3) in *Figure* 7.1). If some of the transformations are either infeasible or unprofitable, the schedulability analyzer will report this fact (feedback (2) in *Figure* 7.1) to the transformation engine, requiring it to undo the transformation. Moreover, if the analyzer cannot find a feasible schedule, it may request more effort to be spent on analyses and transformations, in the sense of [31], to enhance the schedulability of the code.

Finally, the schedulability analyzer outputs certified intermediate code from which the compiler backend will generate executable code. In the current implementation, the C++ compiler and linker is used, as discussed in the following section.

7.7 The Linker

As mentioned, no specific architecture is considered at the moment. The target code machine implementation is a mixture of native C++ statements for some control statement support and a set of C++ class objects, types, and resources for the kernel interface. The linker is simply the C++ compiler. This compiles the certified intermediate code generated by the schedulability analyzer, and links that code with kernel code, as well as with the basic C++ classes.

The executable code generated in this stage is executed by the run-time environment, which simulates distributed processing of the code over a network of processors.

7.8 The Run-time Environment

The run-time environment consists of a kernel, a network simulator and a user interface. It is designed as a single program. The linker combines that program with the application intermediate code. As previously mentioned, the run-time component of the platform is a test environment. No new research ideas are applied to that part of the prototype. This section explains the role of every subcomponent of the run-time environment beginning with the kernel.

7.8.1 The Kernel

Basically, the kernel is a continuous loop. Every iteration, it checks an event list, picks some event, and performs the appropriate action. Events include: scheduling a method/thread, executing a call to a method, sending a message to a remote object (making a call to a method of that object while the object is assigned to a different processor), and updating object queues. Every entry in the event table has a time-stamp to determine when the kernel should react to that event. Every object has a

queue to serialize access to all methods exported by that object. The order in that queue depends on the scheduling criteria used and the arrival order of messages. Typically, there is a kernel for every processor in the network. While, in the current implementation the platform will have only one physical kernel used for emulating a kernel per processor, the platform is scalable to any number of processors and portable to different machines.

There is one master real-time clock for the entire system using abstract real-time units and the kernel is responsible for updating this clock. All events are stamped with time of occurrence. The kernel responds to an event by initiating the required activity; for example, by activating thread execution or initiating the execution of methods. Thus, calls (except some calls to local methods or system libraries) are directed as requests or as events to the kernel. The kernel actually makes the call by executing the callee method. The implementation addresses two call support problems which are inherent in distributed environments: consistency of the values of *out* parameters at the conclusion of the call, when the execution of the caller is resumed, and restoring old state after preemptions. Those problems are addressed when the store-forward mechanism is discussed later in this subsection.

The kernel algorithm is an infinite loop. Every iteration of the loop, the kernel checks the event table sorted by the time and process one event. It first increments the current time by one unit. The kernel interacts with the network simulator to handle messages that have reached their destination. Each message received is decoded, and the kernel updates the corresponding object queue accordingly. Browsing the event table, the kernel selects events with time-stamp equal to the current time. For these selected events, the kernel reacts with the appropriate action, which may be activation of a thread, or sending a message. Sending messages is performed by passing that message to the network simulator which will simulate the propagation of that message to its destination.

The kernel maintains two sets of queues: object queues and processor queues. Access to the object will be serialized using its queue. All requests (calls) to services (methods) provided by this object will be added to its queue. The object queue is a general priority-based queue. Every processor may host multiple objects. The processor queue contains the highest priority requests from the object queues assigned to that processor. Every loop iteration in the kernel algorithm, the object queues of every processor will be checked. If there are any calls still pending, one of them will be scheduled to run. The selection of the method to be executed will be based on some real-time scheduling criterion. In the current implementation, Earliest Deadline First scheduling is used for the sake of testing. However, any scheduling discipline, established or experimental, can be used. The kernel executes the code of that method/thread, which may generate a new set of events. The kernel marks the new events with the correct time-stamp and add them to the event table. The kernel refers to the output of the timing tool to get the static estimate of the execution time. This estimate is used to stamp events produced by the executed method.

As mentioned earlier, the kernel makes the calls to callee object methods. This raises three issues. First, the kernel must remember values of *out* parameters of the call and pass them back to the caller, both for local calls, and for remote calls to methods of other objects assigned to different processors. The issue becomes still harder for remote calls that invoke other calls. The second issue is similar, but arises from preemption of the method. The kernel must remember the values of local variables to resume execution correctly afterward. Finally, the kernel must remember the method program counter, in order to determine the next statement to execute after resuming execution and to keep track of the elapsed time. Note however that these issues, and the transformations used to solve them, will not affect the simulated behavior of the program.

We start by addressing the third issue. Every method/thread is subdivided into a set of non-preemptable units (submethods/subthreads). Every unit then runs to completion without preemption. The criteria used for determining preemption points are based on calls. Whenever a call is found, the method/thread is subdivided into two. The first part ends at the call, while the second part starts with the statement following the call. The second part is further subdivided if another call is found, and so on. A discussion of how the kernel handles the execution of these units is provided later in this section.

To overcome the second issue, the scope of the declaration of local variables defined within a method is changed to be the scope of the object (assuming all recursive calls are unwound). In other words, local variables for any method become part of the object internal state. Variables are renamed, e.g., by using the method name as a prefix, so that no two methods assign a common name incompatibly. Thus, in the case of local calls, the kernel does not worry about *out* parameters, as every variable (including the parameters) are part of the object state and thus can be seen by other methods in the object. This will also hold for those submethods generated by inserting preemption points, as just discussed. *Figure* 7.4 shows the change in code due to insertion of preemption and changing the scopes of local declarations.

For external calls, the solution is quite different, as the caller and callee do not share state. A store-and-forward mechanism, similar to SUPRA-RPC [93], is used instead to remember the parameters of the previous call. For example, if the first call makes another external call, the values of the parameters of the first call need to be retrieved in order to resume execution after returning from the second call. In store-and-forward, the values of input parameters of the caller are usually passed in addition to the parameters required by the callee. Thus, calls to methods have a variable list of parameters. Whenever an external call is found within a method/thread, code must be added to store those parameters. All methods

```
ORIGINAL                          TRANSFORMED

Object O1                         Object O1

  var v1,                           var v1,
  var v2                            var v2
                                  private:
  method m1                             var m1_mv1,
     var mv1,                           var m1_mv2,
     var mv2,                     method m1_1
      .                                 .
      .                                 .
     call O2.m1()                   call O2.m1()
      .                           endmethod m1_1
      .                           method m1_2
      .                                 .
      .                                 .
     call O3.m5()                   call O3.m5()
      .                           endmethod m1_2
      .                           method m1_3
      .                                 .
      .                                 .
      .                                 .
  endmethod m1                    endmethod m1_3
```

Figure 7.4 Example of the insertion of preemption points

and threads will use a standard parameter list consisting of two stacks. The first stack has the parameters of the call. Statements will be added to the code of the method/thread to pop the parameters from that stack. All the parameters of that call will be pushed again onto the stack at preemption points (when making calls) so that they can be retrieved when the call returns. The second stack contains the source object and the next submethod to be executed. The kernel pops that stack when a call returns to determine which object made that call.

Every (non-local) call in the program is replaced by a call to the function store-and-forward. The parameters needed for store-and-forward are: *source id* (where to return), *target id* (which method to call), and *the actual parameters of the caller.* The post-processor replaces external calls with a return statement. The id's referred to above can be addresses or object id and method name. For example: if $O1.m1\_1$ calls $O2.m3$ then the source id will be the address of $O1.m1\_2$, while the target id will be the address of $O2.m3\_1$. *Figure* 7.5 shows an example of the code transformations performed by the post-processor to support store-and-forward: the external call has been replaced with a store-and-forward request to the kernel, and the method returns. Later, the kernel will send a message to the target object and resume execution at $m1\_3$ upon the return from the call to $O2.m3$.

As the motivation for this transformation of the code is to enable the implementation of the run-time kernel, a decision was made to implement it by a post-processor of the intermediate code just before integrating the code with the linker. The input to the post-processor is basically C++ code; the output will also be C++.

The kernel interacts with the other subcomponents of the run-time environment, as shown in *Figure* 7.1. First, it calls the network simulation routine to calculate communication delays through the network when invoking an external call. In addition, the kernel measures the execution time of various threads and methods

```
ORIGINAL

Object O1

  var v1,
  var v2

  method m1
     var mv1,
     var mv2,
      .
      .
     call O2.m3(..)
      .
      .
  endmethod m1
```

```
TRANSFORMED

Object O1

     var v1,
     var v2
  private:
         var m1_mv1,
         var m1_mv2,
  method m1_1
         .
         .
     store_and_forward(O1.m1_2,O2.m3_1,..)
     return
  endmethod m1_1
  method m1_2
         .
         .
  endmethod m1_2
```

Figure 7.5 Example of the application of the store-and-forward mechanism

and reports that to the user along with other statistics through the user interface module, as discussed in *Subsection* 7.8.3. A local time variable is added to every method/submethod. An increment statement for the time variable after every basic block is added to the code generated by the timing tool. The increment reflects the execution time for that basic block. *Figure* 7.6 shows the C++ translation of the method *update_status* in the CRL example of *Subsection* 7.2.5. The static prediction of the execution time of basic blocks by the timing tool is propagated to the kernel by adding a local time variable. This time variable will be incremented at basic block boundaries to accumulate the execution time. The value of time is pushed to the stack and the kernel will pop it to know the execution time of that method (not including any communication delays). The purpose of providing measures of execution time at run-time is to correlate the basic timing measures based on the timing map with actual execution. For some operations, the amount of time is an integer constant, while for others it is expressed as a parameterized expression. In practice, some of these parameterized expressions depend only on compile-time information as operand list lengths, or iteration and time constraint requirements, and are thus easily resolved and specialized into constants statically. However, timing expressions may also depend on the distribution of operands, objects, and processes across the network (this is relevant in calls) and on the usage of shared resources.

In the next section, the network architecture simulator is described.

7.8.2 The Network Architecture Simulation Tool

The network simulation tool provides the timing delay that thread execution anticipates due to distributed allocation of objects. The simulator uses architectural information including a description of the network topology, various distances between nodes, and the transmission medium, as provided in architecture description file.

```
int update_status_1(System_Stack *sp ) {

   long time = 0;

   cin >>  x ;
   time += 3;
   cin >>  y ;
   time += 3;
   cin >>  theta ;
   time += 3;

   // CALL  to vel.get(theta,speed)
   sp->Param_Stack.pushPointer((void*) &theta);
   sp->Param_Stack.pushPointer((void*) &speed);
   sp->Object_Stack.pushPointer((void*) this);
   sp->Object_Stack.pushPointer((void*) &update_status_2);
   sp->Object_Stack.pushPointer((void*) &vel);
   sp->Object_Stack.pushPointer((void*) &vel.get_1);

   sp->Param_Stack.pushLong(time);

   store_forward(self.id,"navigation.update_status_2:navigation",
                 vel.id,"vel.get_1:navigation",sp->no);

   return(1);
}
```

Figure 7.6 An example of the final code to be linked with the kernel

Initially, the simulator reads an assignment file generated by the assignment tool, providing a mapping for every object to a processor. Interaction with the kernel is in the form of requests providing the source object and the target object as well as the size of the message to be sent. The simulator consults the object map, and determines the source processor and the target processor. Using the topology description, it then finds the appropriate route along which to transfer the request.

There is a message queue in every node maintained by the network simulator. If a message is to be transferred on a busy link, it will be queued until the link is free. The transmission rate will be dependent on the medium and the distance the message has to travel. The simulator consults some internal table (data sheet) to calculate the transmission time over that line. The kernel will not block waiting for the results of that request. The results of that call are reported back using the same message format but the previous target object becomes a source for the return. The total communication delay time is the sum of the transmission times and the communication queuing time (forward for the request and backward for the results). The total service time for the kernel request is the sum of the communication delay, the execution time of the specified method within the target object, and the object queuing delay.

There is no interaction between the network simulator and the user interface in the current implementation. All results and status reported to the user come only from the kernel. In the future, a graph may be provided to show the current status of the network, including communication queues and bottlenecks. In the coming section, the user interface subcomponent in the run-time environment is described.

7.8.3 User Interface

In the current implementation the user interface is used only to display measurements and statistics on the applicability of transformations and their effects on performance, deadlines, and processor utilization. Development of a graphical interface is work in progress. It eventually will be possible to draw execution progress figures, providing the user with information on the activities of every processor. Moreover, the measurements and statistics mentioned above will also be presented using graphs. In the future, these capabilities may be extended to include a facility for affecting the execution behavior and for providing run-time assertions.

In the next section, the implementation and integration of the speculative execution transformation within the prototype are illustrated.

7.9 Implementation of the Speculative Execution Transformations

To apply the compiler transformation rules for speculative execution, discussed in *Chapter* 4, it is needed to use the control graph, a data dependence representation and the call graph. The input to the transformer is timed intermediate code. The implementation of the transformation rules follows the following steps:

1. The control graph is browsed trying to find a pattern match, to justify the structural preconditions.

2. If a pattern is found, the dependence preconditions are verified.

3. The call graph is consulted to test all blocking conditions.

4. Safety and profitability of the transformation are justified using the timing preconditions.

5. The code is transformed.

In the first step, the control flow graph is browsed matching patterns that satisfy structural preconditions[3]. Because the CRL language allows only explicit calls, there will be no call in the condition of any *if* statement. We thus extended our analysis to consider cases in which the boolean expression of an *if* statement involves a variable modified by an earlier call. If a pattern is found, the dependence and blocking preconditions are to be verified. The static prediction of the execution time of basic blocks, stored in the control graph, is used to check the safety and profitability of speculative execution according to the timing preconditions. Scanning the control flow graph commences from the top down towards the end and outer to inner in nested constructs. While possible matches within nested compound statements (loops and conditionals) are considered, currently the first feasible match is picked. In the future, we would like to consider alternate opportunities and pick the most profitable one.

The call graph and the object assignment file are used to justify blocking conditions. A call is considered blocking if it is made to a method of an object not assigned to the same processor as the caller. Non-blocking calls are those made to methods of the same object, and which do not invoke any blocking calls. The call graph is consulted to check calls made from the callee method in order to verify the non-blocking nature of the call by checking descendents of the callee. In addition, the assignment file is checked to avoid deadlocks because shadow execution can cause a deadlock if it makes a call to a method on the caller processor, in this case.

As mentioned in *Section* 7.3, pointers in the control graph are maintained to relate every basic block to the corresponding intermediate code generated by the compiler. Currently, these pointers are the starting and ending line numbers

[3]All transformations are applied before doing any changes to the code to enable integration with the run-time environment, as discussed in *Section* 7.8. Thus the control graph will be still reflecting the original structure of the program.

of C++ translation of the basic block. Knowing line numbers facilitates carrying out the action part of the rules (and supports debugging and monitoring). The transformation is performed by creating a new method whose body includes the code to be speculatively executed. The new speculative method will be called from the original code. The values of variables modified in the new speculative method will be saved before making the call and retrieved in case of rollback. The kernel needs to detect the speculative nature of the call and to update the execution time as if the call is performed in parallel with the current execution in the caller. A naming convention is used for the new speculative methods so that the kernel can recognize them. The kernel will not update the clock until validating the speculative execution. *Figures* 7.7 and 7.8 show the transformed version of the timed intermediate code in *Figure* 7.3 after applying the compiler rule of *Figure* 4.6. Note that there is no update of the time after the call to *SPEC_1_action*. The kernel will realize from the name that it is a speculative execution and will store the execution time for correct updating of the clock upon validation of the speculative execution. The kernel library function *update_spec_time* will consider the parallel execution of the *SPEC_1_action* and the remote execution of *abs* and increment the clock with the maximum of their execution times. Note that the transformation will not affect the timing constraints of the block. Currently the speculative execution transformer handle only "no later than" timing constraints. Future extensions include verifying other types of constraints.

After integrating the speculative execution transformer with the other tools, some experiments have been performed to test the applicability and profitability of speculative execution in actual applications. Fragments of a small set of real-time applications have been translated to CRL and tried. In the next section, the results of that experiment are discussed.

```
void SPEC_1_action(GOAL goal,Integer &index) {

  long time=0;

  time += 2;      // time for the condition evaluation
  if ( speed < VHIGH ) {
    action_dtheta = action_rtheta;
    time += 2;
  }
  else {
    this->safeDtheta(action_rtheta,roll,action_dtheta);
    time += 5;
  }
}
```

Figure 7.7 An example of applying speculative execution transformations

7.10 Experimental Results

An experiment was performed to capture the impact of speculative execution on performance of actual real-time applications. Fragments of real-time applications including navigation control, quality monitoring, multi-motor control system, passive sonar, and air traffic control applications, based on the description in [9, 28, 29, 31, 86] respectively, have been translated into CRL. The size of programs and various frequencies of statements are shown in *Figure* 7.9, reflecting, respectively, the number of conditionals, loops, blocking calls, non-blocking calls, input/output, assignment, and other statements including comments and declarations. Each application was compiled and analyzed for static timing behavior. The generated timed intermediate code was linked with the kernel and the run-time performance monitored. Then, the timed intermediate code is reconsidered by the speculative execution transformer linking the output code with the kernel. The new version was executed and the performance was compared with that of the version without speculative execution.

In the experiment, processors are assumed to be homogeneous and interconnected through a bus topology by Ethernet [47] (without repeaters). While

```
void action(GOAL goal , Integer& index ) {

  long time=0;

  this->update_status();
  time += 14;      // time for the call
  time += 2;       // time for the condition evaluation
  if( goal[index].passed ) {
    action_gx = goal[index].x;
    action_gy = goal[index].y;
    index=index + 1 - ((index + 1) / NCOORD) * NCOORD;
    time += 10;
  }
  self->compRelAtt(theta,x,y,action_gx,action_gy,action_rtheta);
  time += 13;
  // save the modified variables
  sp->Param_Stack.pushPointer((Void*) &dtheta);
  time += 1;
  this->SPEC_1_action(goal,index);
  action_abs_rtheta.abs(action_rtheta);
  time += 3; // time for the call
  time += 3;       // time for the condition evaluation
  if( action_abs_rtheta < EPS ) {
    // restore the original values of modified variables
    dtheta = *(Rational *)sp->Param_Stack.popPointer();
    time += 1;
    action_dtheta = 0;
    time += 2;
  }
  else {
    // clean up the stack
    sp->Param_Stack.popPointer();
    update_spec_time(time);
  }
  this->compFlapw(roll,speed,action_dtheta,action_wflap);
  time += 7;
  this->compThrottle(roll,speed,action_dtheta,throttle);
  time += 5;
  cout << "Write to THROT :" << throttle << endl;
  time += 3;
  cout << "Write to FLAP :" << action_wflap << endl;
  time += 3;
}
```

Figure 7.8 An example of applying speculative execution transformations

Program	Size	Cond	Loop	Block	Non-block	I/O	Assign	Other
Navigation Control	204	10	3	3	7	14	14	156
Quality Monitoring	232	3	1	2	7	8	14	193
Multi-Motor Control	783	26	9	2	38	28	69	611
Sonar	504	9	0	1	5	0	32	457
Air Traffic Control	1284	22	3	26	103	12	128	990

Figure 7.9 Statistics for test programs used

the network topology and connection type can be changed, a bus topology and Ethernet connection were selected for consistency with the simulation discussed in the previous chapter. Thus, it is possible to capture the effect of contention on performance measures, something that could not be detected by the simulation. As mentioned earlier, in the current implementation, execution time is based on a map for the basic data types (classes) defined by the CRL language. A static assignment of objects to processors is provided manually (the assignment tool is still under development by other members of the real-time laboratory).

The current status of the prototype imposed on the experiment some limitations which we hope to address in the future. Currently, it is only possible to assign objects to execution nodes, which does not allow shadow execution of part of a method on a different processor. In addition, the currently implemented subset of the CRL language does not directly support loops controlled by conditions other than the number of iterations.

While applying the speculative execution, the number of potential opportunities is reported. If a trial to transform the code fails due to violation of one of the preconditions, the tool reports the cause so that the impact of various conditions on the success rate can be studied. The results of applying the speculative transformation rules to the real-time programs mentioned above are shown in *Figure* 7.10. The table reports the number of possible opportunities, feasible application of the

Program	Trials	Success	Structural	Dependence	Blocking	Timing
Navigation Control	2	1	0	0	0	1
Quality Monitoring	2	0	2	0	0	0
Multi-Motor Control	3	1	2	0	0	0
Sonar	9	0	9	0	0	0
Air Traffic Control	16	3	7	1	4	0

Figure 7.10 Opportunities for speculative execution in test programs

Program	Opportunities	Speedup
Navigation Control	1	3%
Quality Monitoring	0	0%
Multi-Motor Control	1	2%
Sonar	0	0%
Air Traffic Control	3	7%

Figure 7.11 Speedup due to speculative execution of conditions

rules, and number of unsuccessful trials due to violation of a certain precondition. Note that the number of trials is different from the number of conditionals in the applications as *elseif* is not counted.

Generally, the success rate have been noticed to be highly influenced by programming styles. Considerable feasible opportunities have been found in programs that follow a modular or an object oriented style, in which the structural preconditions do not have a dominant effect on the applicability of the transformation rules. Moreover, we see in the use of programmer annotation an interesting possibility that can be tried in the future. In addition, relatively small programs in our experiment are considered. We expect to have more opportunities in large modular programs.

The effect of speculative execution on performance for the above programs is shown in *Figure* 7.11. Speedup is measured as the percentage of the reduction in the average execution time due to speculative execution relative to the execution

time without it. While the performance gains are smaller than that observed in the simulation results in the previous chapter, better results are expected by enabling loop transformations and shadow execution. In addition, the programs considered are small. As the simulation results indicated, the gain will scale with the size of programs. Better performance enhancement are expected for larger programs, due to greater modularity.

In this chapter, an implementation of the speculative execution transformation rules in a platform for developing complex real-time systems has been described. The platform is based on a new real-time programming language called CRL. Fragments of actual real-time applications have been translated into CRL and investigated by the speculative execution transformer for safe and profitable opportunities. The experimental results, consistent with the simulation discussed in the previous chapter, indicate the applicability and profitability of speculative execution in real-time applications. The applicability of speculative execution is highly affected by coding style. Significant numbers of opportunities were found in programs written in object-oriented or modular style. Greater gains in performance are expected for large real-time programs. We believe that the speculative execution transformation can be successful in large applications with modular coding style, in programs written in a language that support while loops with an upper bound on the number of iterations, and in distributed real-time applications supporting remote procedural calls. In addition, we think that the transformation rules can be more applicable when real-time programmers are aware of various preconditions to enable the transformations. For example, C programmers that expect parallelization of their code can access arrays by indices instead of pointers to facilitate parallelization of array access by the compiler. Using object oriented programming methodology and minimizing global variables are examples of coding styles that real-time programmers may consider in

order to enable more safe and profitable opportunities for speculative execution in their programs.

CHAPTER 8

CONCLUSION AND FUTURE WORK

This thesis has addressed the problem of performing safe compiler optimization, parallelization and speculative execution techniques in real-time systems. While naive use of compiler optimization and speculative execution may in general both degrade worst-case performance and complicate timing analyses for distributed real-time systems, we have shown that there are opportunities for safe use of these techniques. We have provided guidelines for identifying such opportunities at compile-time, and applying these in generated code. In addition, we have developed compiler transformation rules for machine-independent compiler optimization, parallelization and speculative execution considering a single real-time process at a time.

We have presented a formal proof of the correctness and safety of the transformation rules for speculative execution. We used temporal logic to verify that the rules preserve the semantics as well as the timeliness of a program. We extended the notion of a state in temporal logic to support reasoning on the contents of the program store.

Through simulation, we have shown that speculative execution can enhance timeliness and performance of real-time programs, demonstrating that the increased compilation time used in analyzing safe opportunities is not wasted. Data dependence has a significant impact on the applicability of speculative execution, which is consistent with the simulation results, but opportunities for speculative execution scale with real-time program sizes. Applicability of speculative execution tends to diminish for large sizes of conditional and while loop body blocks, as well as increased frequency of blocking calls. As expected, shadow execution is always more applicable and profitable on average, since rollback is not required.

In addition, the transformation rules have been implemented in a platform for complex real-time applications in the Real-Time Computing Laboratory at NJIT, which we are using to validate our research ideas. The platform is based on a new object-oriented real-time language. The language and its run-time environment are both being developed at NJIT. The speculative execution transformations have been applied to actual real-time applications. The results of this validation show the applicability and usefulness of speculative execution to real-time systems.

In this chapter, we summarize our current research efforts and suggest future directions to extend our work.

8.1 Future Work

The work presented throughout this thesis can be extended, in our opinion, in many directions. The extensions can be classified into two categories: (i) performing additional experiments and expanding the prototype capabilities, (ii) tackling unsolved technical problems.

8.1.1 Extension of the Tool Support

We would like to extend our prototype and expect to investigate the efficiency of our suggested transformations in additional applications, in co-operation with the sponsors of the real-time computing laboratory at NJIT in industry. We have assumed in this thesis a homogeneous memory; we will extend our model to consider the effects of memory hierarchy on the transformations. We also hope to study the safe application of machine-dependent optimization techniques in real-time systems.

We would like to extend our speculative transformation rules to handle heterogeneous sets of processors and relative timing constraints. We hope to study the applicability of the rules for a specific real-time architecture. Moreover, we may

address the possibility of including instruction-level speculative execution as provided by some architectures.

Currently, we assume abundant resources (processors) which can then be used to speculatively execute part of the code; we hope to relax this restriction in the future. We will investigate the interaction between the transformation engine and assignment tool to provide feedback in the presence of infeasible transformations. We would like to study the possibility of incorporating user assertions or user interaction to guide transformations. In addition, we would like to consider the interaction with various scheduling disciplines and with schedulability analyzers.

Another interesting research direction is to study the impact of phase ordering of real-time compiler transformations. Some transformations may affect the feasibility and usefulness of others. We also hope to build a tool, like the one described in [109] to study the interaction between various real-time compiler transformations.

8.1.2 Work on Technical Problems

Speculative execution can be useful in achieving real-time fault tolerance. Two schemes, *passive* and *active* replication, are commonly used to replicate servers that fail independently. We have proposed a semi-passive architecture for fault tolerance, in which some replicas may be active a fraction of the time to speculatively execution part of the code [117]. We have shown that speculative execution (on a shadow) can enhance overall performance and hence shorten the recovery time in the presence of failure. The compiler is used to detect opportunities for speculative execution, to insert checkpoints, and to construct update messages. We plan to extend the compiler-assisted approach for inserting checkpointing to achieve schedulability-analyzable fault-tolerant real-time systems. In the future, we shall be extending the semi-passive architecture to allow sharing of shadows between multiple

primary processors. In addition, we expect to extend the fault model to handle linked failures and some multiple faults. We also plan to use formal dependability analysis techniques to verify timely recovery.

Speculative execution shares some features with intelligent backtracking in logic programming [12]. We hope to explore the possibility of using similar techniques to minimize the penalty of rollback. In addition, we would like to apply both the analysis of [61], and our transformations, to obtain parallel and speculative execution of code to be executed following an exception, or to speculatively execute ordinary code under the assumption that an exception does not occur. However, care must be taken that ordinary execution in the presence of exceptions will not have a permanent effect.

In this thesis, we formally verified that the speculative execution transformation rules preserve semantics and timeliness. We plan to formally specify and verify various other code transformations, and hope to extend that approach to a general real-time compiler transformation specification and verification tool, like that of [109]. In addition, we would like to tackle the much tougher problem of formal verification of transformations in distributed real-time systems.

Real-time compiler transformations can be very useful in enabling non-intrusive activities such as monitoring and debugging without affecting timing constraints. In safety-critical applications, simulation is usually used to test the code, since it is impossible to develop the code on the target environment due to high risk and cost. Gains in performance by optimization can be replaced by delays. Using inserted delays as a placeholder for debugging and monitoring, it is possible to capture more accurately the behavior of programs in the target environment. In addition, bugs may be isolated more easily. We would like to investigate the effectiveness of compiler optimization in supporting non-intrusive monitoring and debugging.

Currently, we are working on the problem of applying compiler optimization techniques to distributed real-time systems [114]. We are extending our analysis to consider the effect of compiler optimization and speculative execution transformation in one process on the other processes in the system (multiprocess analysis). We have addressed the difficulties associated with performing compiler optimization in distributed real-time systems, and developed an algorithm to apply machine-independent code improvement optimization safely in such a distributed environment. The algorithm uses resources' busy-idle profiles [36] to investigate effects of optimizing one process on other processes, where a restricted form of resource contention [97] is assumed to simplify analysis. In the future, we plan to extend the resource contention model to allow for resource optimization and handle nested calls to shared resources.

We believe that our research enhances the confidence of real-time systems programmers in high-level language development, and allows them to rely on compiler optimization. Our study provides guarantees for safe application of compiler optimization and parallelization techniques. We believe that the studies we are conducting will be essential for the design and development of complex real-time systems. The development of such systems will require the assistance of compiler optimization techniques to tune performance and enhance resource utilization without destroying the timing behavior of the system. In addition, currently running real-time applications can still benefit, by being recompiled to enhance their response time which increase their robustness and reliability.

APPENDIX A

COMPILER OPTIMIZATION SUPPORTING ANALYSIS

Code optimization can be divided into three interrelated areas. *Local optimization* is performed within a basic block of code. A basic block is a sequence of consecutive statements which may be entered only at the beginning and when entered is executed in sequence without halt or possibility of branch except at the end [2]. *Loop optimization* is a transformation of code in a loop, e.g., lifting invariant statements or strength reduction of calculations. *Global optimization* is supported by data flow analysis – the determination at compile-time of information giving facts about communication and use of data. Data flow analysis can be seen as the transmission of useful relationships from all parts of the program to the places where the information can be of use. Data flow analysis includes intraprocedural analysis – analysis of a single function or procedure – and interprocedural (interprocess) analysis [60].

In this appendix, the three forms of code optimization are elaborated, as well as other forms of analysis used to enable optimization. The discussion begins with control flow analysis, followed by representation of basic blocks. Next, the application global data flow analysis is shown. This appendix is concluded by a brief discussion of some machine-related optimization techniques.

A.1 Control Flow Analysis

Control flow analysis is the determination of the structure of a program. It identifies possible execution paths as well as basic blocks within the program. Control flow analysis enables application of local optimization techniques by breaking the code into basic blocks. The basic blocks and their successor relationships are often represented as a directed graph called a *flow graph*. Nodes of the flow graph are basic

blocks, and edges represent control flow. A loop has a single entry node, and the nodes in the loop body form a strongly connected region. As an example of the use of control flow analysis, consider a conditional statement with identical code in its *then* and *else* branches. It may be possible to optimize the size of the program by *hoisting* the common code before the conditional (*Figure* A.1). A related optimization, using the flow graph during instruction scheduling, can improve performance if execution is pipelined by filling the conditional delay slot.

```
   ORIGINAL                    OPTIMIZED

if (s1)                      s2;
  s2;                        if (s1)
  s3;                         s3;
else                        else
  s2;                          s4;
  s4;                       endif
endif                       s5;
s5;
```

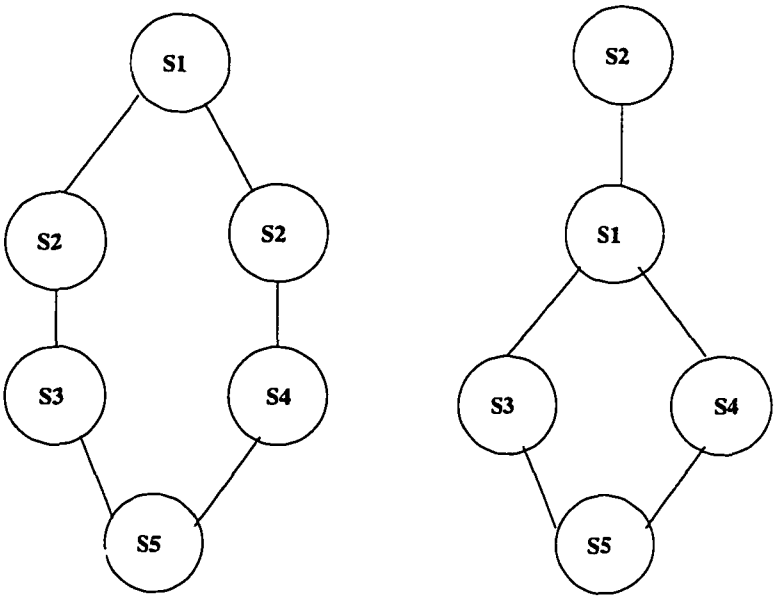

Figure A.1 Code hoisting

A.2 DAG Representation of Basic Blocks

A directed acyclic graph (DAG) representation is commonly used to automatically analyze code in a basic block, giving a picture of how values computed by one statement in the block are used in others. Constructing the DAG allows determination of common subexpressions, names with external reaching definitions, and names or expressions whose values may be used outside the block. Local optimization, such as common subexpression elimination, and copy and constant propagation, can be applied within the block. Consider for example, the basic block in *Figure* A.2.

```
   ORIGINAL                    OPTIMIZED

[S1]  c := a+b              [S1]   c := a+b
[S2]  d := a+b              [S2']  d := c
```

Figure A.2 Common subexpression elimination

There is no need to compute the expression `a+b` twice, since `a` and `b` do not change. Therefore the value of `c` can be used. Furthermore, the use of `d` can be replaced with `c`, using copy propagation.

A.3 Global Data-flow Analysis

A number of optimizations can be realized by comparing various pieces of information which can only be obtained by examining the entire program. For example, if a variable A has the value 3 every time control reaches a certain point p, then 3 can be substituted for each use of A at p. This gathering of information from the entire program occurs via *global data-flow analysis*. Global data-flow analysis relates the definition of variable and constants with their uses throughout the program.

One standard data flow problem is *Reaching Definitions*, that is, the problem of which definition of a variable can possibly reach a given program point. In the previous example, it is necessary to know which values A might have when reaching p. So, two sets are computed for each basic block. The first is the *Gen* set of generated definitions. The second is the *Kill* set of defined identifiers redefined in the block. The basic block DAG can be used to generate those sets, and those sets can be used to compute the IN and OUT sets of defined identifiers.

A.4 Intraprocedural and Interprocedural Analysis

Data-flow analysis generally pertains to relationships among the definitions and uses of variables occurring in the program. An *intraprocedural* program analysis considers an individual procedure in isolation from the rest of the program. In the spirit of separate compilation, it is assumed during the analysis of procedure P that information about the program outside the boundaries of P is not available. An *interprocedural* program analysis takes place across procedure boundaries. During the analysis of P, the results of analyzing other procedures are utilized. Consider constant propagation. If no knowledge is available or utilized regarding values of formal parameters and globals on entrance to, or return from, procedures, then the constant propagation analysis is intraprocedural. An interprocedural constant propagation algorithm [23] improves this knowledge in particular by attempting to recognize when a formal parameter always has the same value upon entrance to a procedure, and incorporating this information into the propagation of constants within the procedure. For another example, consider the code in *Figure* A.3:

If it is known that neither a nor b will be modified in P2, the second expression $a + b$ can eliminated.

```
Procedure P1()
   .
   .
 x := a + b;
 call P2(*a, *b)
 y := a + b;
   .
   .
```

Figure A.3 An interprocedural constant analysis

Some interprocedural problems can be solved using only the function call graph of a program; others require more sophisticated representations combining interprocedure and intraprocedural flow, such as the technique presented in [54]. Meanwhile, explicitly parallel programs often need representation with multiple classes of edges, as discussed in [62].

A.5 Peephole Optimization

Peephole optimization is a technique used in many compilers in connection with the optimization of either intermediate or object code. It occurs in the compiler back-end, during code generation. Peephole optimization works by looking at the intermediate or object code within a small range of instructions (a peephole), although the code in the peephole need not be contiguous. It is the characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements; thus, repeated passes over the code may be necessary to get maximum benefit. Peephole optimization includes removal of redundant loads and stores, detection of unreachable code, and simplification of multiple jumps, as well as other machine-related optimization like reduction in strength, use of machine idioms, and efficient register usage.

APPENDIX B

RULES FOR MACHINE-INDEPENDENT OPTIMIZATION

RULE: DAG Optimization (Copy Propagation)

Preconditions:
Structural:
(1) $S = S_1S_2S_3$; where S_1: $a = exp1$, S_3: $b = exp1$.
(2) *exp1* does not contain any critical sections or access to a shared resource.
Dependence:
(3) S_2 affects neither a nor the variables in *exp1*.
(4) There is no dependence from S_2 to S_3.
Timing:
(5) S will meet its deadline.
(6) *exp1* contains at least one memory stored variable or a is a register variable and *exp1* is not a constant.

Action:
Transform S_3 into
S_3: $b = a$.

Postcondition:
No deadline will be missed.

RULE:	Code Hoisting
Preconditions:	Structural: (1) $S = S_1$; if (exp) then $S_2 S_e S_2'$ else $S_3 S_e S_3'$. (2) Neither S_2 nor S_3 contains any critical sections or access to a shared resource. Dependence: (3) There is no dependence from S_2 or S_3 to S_e. (4) There is no data or resource dependence from exp to S_e. Timing: (5) Either branch will meet its deadline.
Action:	Transform S into S_e (if (exp) then $S_2 S_2'$ else $S_3 S_3'$)
Postcondition:	No deadline will be missed.
Comment:	will not interfere with dependences from S_e to S_2' or S_3' since otherwise would have been output or resource dependences from S_2 or S_3 to S_e.

RULE:	Code Sinking
Preconditions:	Structural: (1) $S = S_1$; if (exp) then $S_2 S_e S_2'$ else $S_3 S_e S_3'$. (2) Neither S_2' nor S_3' contains any critical sections or access to a shared resource. Dependence: (3) Neither S_2' nor S_3' depends on S_e . Timing: (4) Either branch will meet its deadline.
Action:	Transform S into (if (exp) then $S_2 S_2'$ else $S_3 S_3'$) S_e
Postcondition:	No deadline will be missed.
Comment:	Will ordinary prefer to use Code Hoisting when both are applicable.

RULE: Common Subexpression Elimination #1

Preconditions: Structural:
(1) $S = S_1S_2S_3$; where S_1 and S_3 compute the same expression.
Dependence:
(2) S_3 does not depend on S_2.
(3) S_2 may depend on S_1 (i.e. no jump from S_1 to S_3).
Timing:
(4) S will meet its deadline.

Action:
Transform S into S_1S_2 (references to S_3 use S_1 instead).

Postcondition:
No deadline will be missed.

Comment:
There cannot be an output or resource dependence from S_1 to S_2, since otherwise there would have be one from S_2 to S_3.

RULE: Common Subexpression Elimination #2

Preconditions: Structural:
(1) $S = S_1S_2S_3$; where S_1 and S_3 compute the same expression.
Dependence:
(2) S_2 does not depend on S_1 .
Timing:
(3) S will meet its deadline.

Action:
Transform S into S_2S_3 (references to S_3 use S_1 instead).

Postcondition:
No deadline will be missed.

Comment:
Case of dead code elimination.

RULE: Common Subexpression Elimination #3

Preconditions: Structural:
(1) $S = S_1S_2S_3$; where S_1 and S_3 compute the same expression.
(2) S_2 does not contain any critical sections or
access to a shared resource.
Dependence:
(3) There is a true dependence from S_2 to S_1.
(4) S_2 may depend on S_1 (i.e. no jump from S_1 to S_3).
Timing:
(5) S will meet its deadline.
(6) S_3 uses at least one memory variable.

Action:
Transform S into
$S_1(a = value)S_2$ (use a) (refernces to S_3 use of the value of a).

Postcondition:
No deadline will be missed.

RULE: Copy and Constant Propagation

Preconditions: Structural:
(1) $S = S_1S_2S_3$;
where S_1 : $a = b$, b is a live variable or constant
Dependence:
(2) Neither a nor b is redefined in S_2 .
(3) a is dead at S_3 .
Timing:
(4) Both a and b are of homogeneous type or b is a register
variable (or constant if immediate instructions are at least
as fast as register instructions).
(5) S will meet its deadline.

Action:
Transform S_2 into S_2' by replacing every use of a with b
Transform S into S_2' S_3 .

Postcondition:
No deadline will be missed.

RULE:	Dead Assignment Elimination
Preconditions:	Structural: (1) $S = S_1S_2S_3$; where S_1 is $a = exp$ Dependence: (2) no other segments depend on a . Timing: (3) S will meet its deadline.
Action:	Remove S_1
Postcondition:	No deadline will be missed.

RULE:	Dead Branch Elimination
Preconditions:	Structural: (1) $S = S_1$; where S_1 is if (exp) S_2 else S_3 (2) exp is always true. Timing: (3) S will meet its deadline.
Action:	Transform S into S_3 .
Postcondition:	No deadline will be missed.
Comment:	Symmetric case for exp always false.

RULE:	Dead Conditional Elimination
Preconditions:	Structural: (1) $S = S_1$; where S_1 is if (exp) S_2 else S_3 (2) S_2 and S_3 both are empty. (3) exp contains neither an assignment nor a resource use. Timing: (4) S will meet its deadline.
Action:	Remove S .
Postcondition:	No deadline will be missed.

RULE: Reduction in Strength

Preconditions:
Structural:
(1) $S = S_1$; where S_1 is on the form $a = constant * b$
(2) S is not inside a loop.
Timing:
(3) S will meet its deadline.

Action:
Transform S into $a = b + b + b + ...constant$ times.

Postcondition:
No deadline will be missed.

Comment:
Require that repetitive addition to be less time-consuming than multiplication (highly machine dependent).

RULE: Reduction in Strength in a loop #1

Preconditions:
Structural:
(1) $S = S_1S_2S_3S_4S_5$; where
S_1 is the segment before the loop,
S_2 is the beginning of the loop,
S_3 is of the form $a = const_1$ * i + $const_2$ with i loop index,
S_4 is the rest of the iteration body,
S_5 is the next segment after the loop.
(2) S_4 contains critical section call.
(3) The number of iterations > 1.
Dependence:
(4) The only definition of i in S_4 is the increment statement.
Timing:
(5) S will meet its deadline.

Action:
Define S_1' : $a = const_1$ * i initial + $const_2$.
Define S_4' : S_4 substituting i with i initial.
Define S_3' : $a = a + const_1$.
Define S_2' : S_2 substituting initial with next(initial).
Transform S into $S_1S_1'S_4'S_2'S_3'S_4S_5$

Postcondition:
No deadline will be missed.

Comment:
Highly machine dependent.

RULE: Reduction in Strength in a loop #2

Preconditions:

Structural:

(1) $S = S_1S_2S_3S_4S_5$; where

S_1 is the segment before the loop,

S_2 is the beginning of the loop,

S_3 is of the form a = $const_1$ * i + $const_2$ with i loop index,

S_4 is the rest of the iteration body,

S_5 is the next segment after the loop.

(2) S_4 does not contain critical section call.

(3) The number of iterations > 1.

Dependence:

(4) The only definition of i in S_4 is the increment statement.

Timing:

(5) S will meet its deadline.

Action:

Define S_1' : a = $const_1$ * i initial + $const_2$.

Define S_3' : a = a + $const_1$.

Transform S into $S_1S_1'S_2S_4S_3'S_5$

Postcondition:

No deadline will be missed.

Comment:

Highly machine dependent.

RULE: Invariant Code Motion

Preconditions: Structural:

(1) $S = S_1S_2S_3S_4S_5S_6$; where
  S_1 is the segment before the loop,
  S_2 is the header of the loop,
  S_3,S_5 is loop variant code,
  S_4 is loop invariant code,
  S_6 is the next segment after the loop.

(2) $S_2S_3S_4S_5$ will be executed at least once.

(3) S_3 does not contain critical section call.

(4) S_5 may contain critical section call.

Dependence:

(5) S_4 does not depend, directly or transitively, on the loop index.

(6) S_4 does not depend, directly or transitively, on any resource use.

(7) Both S_3 and S_5 depend on the loop index.

Timing:

(3) S will meet its deadline.

Action:

Transform S into $S_1S_4S_2S_3S_5S_6$

Postcondition:

No deadline will be missed.

RULE: Invariant Code Peeling

Preconditions:
Structural:
(1) $S = S_1S_2S_3S_4S_5S_6$; where
S_1 is the segment before the loop,
S_2 is the header of the loop,
S_3,S_5 is loop variant code,
S_4 is loop invariant code,
S_6 is the next segment after the loop.
(2) $S_2S_3S_4S_5$ will be executed at least once.
(3) S_3 may contain critical section call.
(4) S_5 may contain critical section call.
Dependence:
(5) S_4 does not depend, directly or transitively, on the loop index.
(6) S_4 does not depend, directly or transitively, on any resource use.
(7) Both S_3 and S_5 depend on the loop index.
Timing:
(3) S will meet its deadline.

Action:
Define S_3' : S_3 substituting i with i initial.
Define S_5' : S_5 substituting i with i initial.
Define S_2' : S_2 substituting initial with next(initial).
Transform S into $S_1S_3'S_4S_5'S_2'S_3S_5S_6$

Postcondition:
No deadline will be missed.

RULE: Dead Loop Elimination

Preconditions:
Structural:
(1) $S = S_1$; where S_1 is loop (exp) S_2
(2) S_2 is empty.
(3) exp contains neither an assignment nor a resource use.
Timing:
(4) S will meet its deadline.

Action:
Remove S .

Postcondition:
No deadline will be missed.

APPENDIX C

RULES FOR SPECULATIVE EXECUTION

RULE: SPECULATIVE_IF

Preconditions:

Structural:
(1) $S = (\ if\ (C)\ then\ S_2\ else\ S_3\ )$ is a single-exit code region.
(2) C is a call being executed on another processor
Dependence:
(3) $Vars(S_2) \cap Mod(C) = \phi$
(S_2's variables have correct values immediately before if)
Blocking:
(4) There are no blocking constructs in S_2.
(5) For all methods M in $TCalls(Calls(S_2))$, $not(Blocking(M))$.
(6) For each method M in $TCalls(C) \cap TCalls(Calls(S_2))$, $not(Ordered(M))$.
(Incorrectly or prematurely executing any such statement has a permanent and invalid effect on the environment.)
Timing:
(7) $t_s(Mod(S_2)) + t_f + t_j < Time(C)$.
(Useful work can be done.)
(8) $Time(S_3) + t_r(Mod(S_2)) \leq Time(S_2)$.
(Worst-case time does not increase.)

Actions:

Execute C in parallel with the following:
$save(Mod(S_2))$; S_2.
Insert synchronization between $exit(C)$ and $exit(S_2)$.
Check x_c, the return parameters of C;
If this enables S_2, do nothing.
Otherwise, execute $restore(Mod(S_2))$; S_3.
In any case, continue executing from $exit(S)$.

Postcondition:

S has completed without missing its deadline.
State is as if execution had been sequential.

Comment:

A symmetric rule exists for S_3.

Properties:

Preserves the program semantics.
Does not extend the worst-case execution path.

RULE: SPECULATIVE_WHILE

Preconditions:

Structural:
(1) S = (*while* (C) *do* S_2) is a single-exit code region.
(2) C is a call being executed on another processor
(3) The loop will be executed at least once.
Dependence:
(4) $Vars(S_2) \cap Mod(C) = \phi$
(S_2's variables have correct values immediately before *while*)
Blocking:
(5) There are no blocking constructs in S_2.
(6) For all methods M in $TCalls(Calls(S_2))$, $not(Blocking(M))$.
(7) For each method M in $TCalls(C) \cap TCalls(Calls(S_2))$, $not(Ordered(M))$.
Timing:
(8) $t_r(Mod(S_2)) + t_s(Mod(S_2)) + t_f + t_j < Time(C)$.
(Useful work can be done; no increase in worst-case time.)
(9) $t_r(Mod(S_2)) \leq Time(S_2)$.
(Given at least one iteration; no increase in worst-case time.)

Actions:

Execute C in parallel with the following:
$save(Mod(S_2))$; S_2.
Insert synchronization between $exit(C)$ and $exit(S_2)$.
Check x_c, the return parameters of C;
If this enables S_2, repeat.
Otherwise, execute $restore(Mod(S_2)); exit(S)$.

Postcondition:

S has completed without missing its deadline.
State is as if execution had been sequential.

Properties:

Preserves the program semantics.
Does not extend the worst-case execution path.

RULE: SHADOW_IF

Preconditions:

Structural, Dependence, Blocking Constraints are same as speculative_if rule.
Timing:
(7) $t_c(Mod(S_2)) + t_f + t_j < Time(C)$.
(Useful work can be done; no increase in worst-case time.)
(8) $t_c(Mod(S_2)) + t_f + Time(S_3) \leq Time(S_2)$.
(Worst-case time does not increase along the else branch.)

Action:

Execute C in parallel with the following:
$copy(Mod(S_2))$; $shadow(S_2)$.
When C finishes, check the return parameters of C;
If true, wait (if necessary) for S_2 to complete,
and execute $copy(Mod(shadow(S_2)))$.
Otherwise enables S_3, interrupt the shadow execution of S_2,
and begin execution of S_3.
In any case, continue executing from $exit(S)$. Postcondition:
S has completed without missing its deadline.
State is as if execution had been sequential. Comment:
A symmetric rule exists for S_3.

Properties:

Preserves the program semantics.
Does not extend the worst-case execution path.

RULE: SHADOW_WHILE

Preconditions:

Structural, Dependence, Blocking Constraints are same as speculative_while rule.
Timing:
(7) $t_c(Mod(S_2)) + t_f + t_j < Time(C)$.
(Useful work can be done.)
(Given at least one iteration; no increase in worst-case time.)

Action:

Execute C in parallel with the following:
$copy(Mod(S_2))$; $shadow(S_2)$.
When C finishes, check the return parameters of C;
If true, wait (if necessary) for S_2 to complete,
and execute $copy(Mod(shadow(S_2)))$.
Otherwise interrupt the shadow execution of S_2.
In any case, continue executing from $exit(S)$.

Postcondition:

S has completed without missing its deadline.
State is as if execution had been sequential.

Properties:

Preserves the program semantics.
Does not extend the worst-case execution path.

REFERENCES

1. M. Abadi and L. Lamport, "An Old-Fashioned Recipe for Real Time," *Research Report 91*, Digital Equipment Corporation, System Research Center, October, 1992.

2. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachussetts, 1986. ISBN 0-201-10088-6.

3. A. Aiken, J. H. Williams and E.L. Wimmers, "Safe: A Semantic Technique for Transforming Programs in the Presence of Errors," *ACM Transaction of Programming Languages and Systems*, Vol. 17, No. 1, pp. 63–84, January 1995.

4. N. Alewine, W. Fuchs and W. Hwu, "Application of Compiler-assisted Rollback Recovery to Speculative Execution Repair," *Proceedings of the Conference on Hardware and Software Architectures for Fault Tolerance Experiences and Perspectives*, Le Mont Saint Michel, France, June 1993.

5. M. Alexander et al., "Memory Bandwidth Optimizations for Wide-Bus Machines," *Hawaii International Conference on Systems and Software (HICSS) 26*, pp. 466–475, 1993.

6. R. Alur and T. Henzinger, "Logics and Models of Real-Time: A Survey," *Lecture Notes on Computer Science 600, Real-time: Theory in Practice*, pp. 74–106.

7. S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI'93)*, pp. 126–138, Albuquerque, New Mexico, June 1993.

8. H. Ando et al., "Speculative Execution and Reducing Branch Penalty on a Superscalar Processor," *IEICE Transactions on Electronics*, Vol E76-C, Vol. 7, pp. 1080–1093, July 1993.

9. D. Auslander and C. Tham. *Real-time software for control : program examples in C*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

10. J. Backus, J. H. Williams and E.L. Wimmers, "The FL Language Manual," *Technical Report, RJ 5339(54809)*, IBM Corp., Armonk, New York.

11. S. K. Baruah and L. E. Rosier, "Limitations Concerning On-line Scheduling Algorithms for Overloaded Real-Time Systems," *Proceedings of the IEEE/IFAC Real-Time Operating Systems Workshop*, Atlanta, Georgia, May 1991.

12. M. Bruynooghe and L. Pereira, *Deductive Revision by Intelligent Backtracking.* Ellis Horwood Publication, Distributed by John-Wiley & Sons, New York, 1984.

13. M. Benitez and J. Davidson, "The Advantage of Machine-Dependent Global Optimization," *Proceeding of International Conference on Programming Languages and System Architectures*, pp. 105–124, Springer-Verlag LNCS, March 1994.

14. M. Benitez and J. Davidson, "A Retargetable Integrated Code Improver," *Technical report No. CS-93-64*, Department of Computer Science, University of Virginia at Charlottesville, November 1993.

15. A. Berstein and P. Harter, "Proving Real Time Properties of Programs with Temporal Logic," *Proceedings of the Eighth Symposium on Operating Systems Principles*, pp. 1–11, 1981.

16. A. Bestavros and S. Braoudakis, "SCC-ns: A Family of Speculative Concurrency Control Algorithms for Real-Time Databases," *Proceedings of Third International Workshop on Responsive Computer Systems*, Lincoln, New Hampshire, September 1993.

17. A. Bestavros and S. Braoudakis, "Timeliness via Speculation for Real-Time Databases," *Proceedings of* 15^{th} *IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December, 1994.

18. A. Bestavros and S. Braoudakis, *Value-cognizant Speculative Concurrency Control.* Proceedings of VLDB'95: The International Conference on Very Large Databases, Zurich, Switzerland, September 1995.

19. A. Broggi, *A Novel Approach to Lossy Real-Time Image Compression: Hierarchical Data Reorganization on a Low-cost Massively Parallel System.* Journal of Real-Time Imaging, Academic Press, New York, (to appear).

20. R. Bringmann et al., "Speculative Execution Exception Recovery Using Write-back Suppression," *Proceedings of the* 26^{th} *Annual International Symposium on Microarchitecture*, pp. 214–223, 1993.

21. P. Chang, et al., "Three Architecture Models for Compiler-Controlled Speculative Execution," *IEEE Transactions on Computers*, Vol. 44, No. 4, pp. 481–494, April 1995.

22. T. M. Chung and H. G. Dietz, "Language Constructs and Transformation for Hard Real-time Systems," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, La Jolla, California, June 1995.

23. D. Callahan, K. Cooper, K. Kennedy and L. Torczon, "Interprocedural constant propagation," *Proceedings of ACM SIGPLAN*, pp. 152–161, June 1986.

24. J. Collard, "Automatic Parallelization of *while*-Loops Using Speculative Execution," *International Journal of Parallel Programming*, Vol. 23, No. 2, 1995.

25. M. Colnarič and W. A. Halang, "Architectural Support for Predictability in Hard Real Time Systems," *IFAC Control Engineering Practice*, Vol. 1, No. 1, pp. 51–57, 1993.

26. P. Cousot and R. Cousot, "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proceedings of ACM SIGPLAN*, pp 238–252, January 1977.

27. J. Davidson and A. Jinturkar, "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses," *Proceeding of the ACM SIGPLAN '94 Conference on Programming Languages Design and Implementation (PLDI'94)*, pp. 186–195, June 1994.

28. S. Davidson, I. Lee, and V. Wolfe, "Time Atomic Commitment", IEEE Transactions on Computers. Vol. 40, No. 5 , pp. 573-583, May 1991.

29. T. Davis, H. Erzberger and S. Green, "Design and Evaluation of an Air Traffic Control Final Approach Spacing Tool," *Journal of Guidance, Control and Dynamics*, Vol. 14, No. 4, pp. 848–854, July 1991.

30. R. Gerber and S. Hong, "Semantics-Based Compiler Transformations for Enhanced Schedulability," *Proceedings of the* 14^{th} *IEEE Real-Time Systems Symposium*, pp. 232–242, December 1993.

31. R. Gerber and S. Hong, "Compiling Real-Time Programs with Timing Constraints Refinement and Structural Code Motion," *IEEE Transactions on Software Engineering*, Vol. 21, No. 5, May 1995.

32. C. Giardina and E. Dougherty, *Morphological Methods in Image and Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

33. A. P. Goldberg, "Optimistic Algorithms for Distributed Transparent Process Replication," Ph.D. Thesis, University of California at Los Angeles, California, 1991.

34. B. Goldberg, "Detecting Sharing of Partial Applications in Functional Programs," *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pp. 408–425, 1987.

35. P. Gopinath and R. Gupta, "Applying Compiler Techniques to Scheduling in Real-time Systems," *Proceedings of the* 11^{th} *IEEE Real-Time Systems Symposium*, pp. 247–256, Orlando, Florida, December 1990.

36. R. Gupta and M. Spezialetti, "Busy-Idle Profiles and Compact Task Graphs: Compile-time Support for Interleaved and Overlapped Scheduling of Real-Time Tasks," *Proceedings of the* 15^{th} *IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

37. R. Gupta and P. Gopinath, "Correlation Analysis Techniques for Refining Execution Time Estimates of Real-Time Applications," *Proceedings of the* 11^{th} *IEEE Workshop on Real-Time Operating Systems and Software*, pp. 54–58, Seattle, Washington, May 1994.

38. W. Halang and A. Stoyenko, *Constructing Predictable Real-Time Systems.* Kluwer Academic Publishers, Boston, 1991.

39. R. Halstead, *Design Requirements for Concurrent Lisp Machines.* McGraw-Hill, New York, pp. 69–105, 1989.

40. P. K. Harter, Jr., "On the Application of Temporal Logic to the Verification of Real-Time Programs," Ph.D. Thesis, Computer Science Department, University of New York at Stony Brook, New York, 1982.

41. C. Healy, D. Whalley and M. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proceedings of the* 16^{th} *IEEE Real-Time Systems Symposium*, pp. 288–296, December 1995.

42. T.A. Henzinger, Z. Manna, and A. Pnueli, "Temporal Proof Methodologies for Real-Time Systems," *Proceedings of the* 18^{th} *Annual ACM Symposium on Principles of Programming Language*, pp. 353–366, 1990.

43. C. A. R. Hoare, *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, New Jersey, 1985.

44. S. Hong and R. Gerber, "Scheduling with Compiler Transformations: The TCEL Approach," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software*, pp. 80–84, May 1993.

45. S. Hong and R. Gerber, "Compiling Real-Time Programs into Schedulable Code," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 28(6):166-176, ACMPRESS, June 1993.

46. T. Huang and J. Liu, "Predicting the Worst-Case Execution Time of the Concurrent Execution of Instruction and Cycle-Stealing DMA I/O Operations," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, June 1995.

47. *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.* Institute of Electrical and Electronics Engineers, ANSI/IEEE standard 802.3, 1990.

48. D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, July 1985.

49. M. Katz, "Optimistic Concurrency with Rollback: an Alternative When Compile-time Parallelization Fails," *Proceedings of the Workshop on Parallelism in the Presence of Pointers and Dynamically-Allocated Objects. Supercomputing Research Center of the Institute for Defense Analyses*, March 1990.

50. K. Kennedy and K. S. McKinley, "Loop Distribution with Arbitrary Control Flow," *Proceedings of Supercomputing '90*, pp. 407–416, November 1990.

51. E. Kligerman and A. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 9, pp. 940–949, September 1986.

52. A. Kral, "Improving Semi-static Branch Prediction by Code Replication," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pp. 97–105, ACMPRESS, June 1994.

53. L. Lamport, "Time, Clocks, and the Ordering of Events in Distributed Systems," *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, July 1978.

54. W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Pointer-induced Aliasing," *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI'92)*, pp. 235–248, 1992. Published as SIGPLAN Notices, 27 (7).

55. J. Lehoczky, "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines," *Proceedings of the* 11^{th} *IEEE Real-Time Systems Symposium*, pp. 201–209, December 1990.

56. Y. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, June 1995.

57. D. Leinbaugh and M. Yamini, "Guaranteed Response Times in a Distributed Hard Real Time Environment," *IEEE Transactions on Software Engineering*, Vol. 12, No. 12, pp. 1139–1144, December 1986.

58. C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol. 20, pp. 46–61, 1973.

59. T. Marlowe and S. Masticola, "Safe Optimization for Hard Real-Time Programming," *Second International Conference on Systems Integration*, Special Session on Real-Time Programming, pp. 438–446, June 1992.

60. T. Marlowe and B. Ryder, "Properties of Data Flow Frameworks: A Unified Model," *Acta Informatica*, Vol. 28, 1990.

61. T. Marlowe, A. Stoyenko, S. Masticola and L. Welch, "Schedulability-Analyzable Exception Handling for Fault-Tolerant Real-Time Languages," *Journal of Real-Time Systems*, Vol 7, pp. 183–212, 1994.

62. S. Masticola, "Static Detection of Deadlocks in Polynomial Time," Ph.D. Thesis, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, 1993.

63. H. Meske, M. Younis and W. Halang, "A Reduced Instruction Set for a Hard Real Time Architecture," *Proceedings of ACM SIGARCH Workshop on Architectures for Real Time Applications (ISCA 94)*, Chicago, Illinois, April 1994

64. S. P. Midkiff and D. A. Padua, "Issues in the Optimization of Parallel Programs," *Proceedings of the International Conference on Parallel Processing*, Vol. II, 105–113, 1990.

65. A. K. Mok, P. Amerasinghe, M. Chen and K. Tantisrivat, "Evaluating Tight Execution Time Bounds of Programs by Annotations," *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, Pittsburgh, Pennsylvania, pp. 74–80, May 1989.

66. A. Mok and M. Dertouzos, "Multiprocessor Scheduling in a Hard-Real-Time Environment," *Proceedings of the Seventh Texas Conference on Computing Systems*, November 1978.

67. F. Mueller, "Compiler Support for Software-Based Cache Partitioning," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, June 1995.

68. F. Mueller, D. Whalley and M. Harmon, "Predicting Instruction Cache Behavior," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, Orlando, Florida, June 1994.

69. K. T. Narayana and A. A. Aaby, "Specification of Real-Time Systems in Real-Time Temporal Interval Logic," *Proceeding of the* 9^{th} *IEEE Real-Time Systems Symposium*, pp. 86–95, December 1988.

70. K. Narasimhan and K. Nilsen, "Portable Execution Time Analysis for RISC Processors," *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

71. V. Nirkhe and W. Pugh, "Partial Evaluation of High-level Imperative Languages, with Applications in Hard Real-Time Systems," *Conference Record of the Nineteenth Annual ACM Conference on the Principles of Programming Languages*, pp. 269–280, 1992.

72. V. Nirkhe and W. Pugh, "A Partial Evaluator for the Maruti Hard-Real-Time System," *Journal of Real-Time Systems*, Vol. 5, No. 1, pp. 13–30, March 1993.

73. J. Ostroff, "Verifying Finite State Real-Time Discrete Event Processes," *Proceedings of the 9^{th} International Conference of Distributed Computing Systems*, pp. 207–216, 1989.

74. S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica*, Vol. 6, pp. 319–340, Springer Verlag, 1976.

75. S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *Technical Report*, Department of Computer Science, Stanford University, October 1980.

76. R. Paige, "Symbolic Finite Differencing," *Proceedings of the Third European Symposium on Programming*, pp. 36–56, 1990 (available also in Number 432 in Lecture Notes on Computer Science, 1990).

77. R. Paige and S. Koenig, "Finite Differencing of Computable Expressions," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 403–454, July 1982.

78. C. Park, "Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths," *Journal of Real-Time Systems*, Vol. 5, No. 1, pp. 31–62, March 1993.

79. A. Pnueli and E. Harel, "Applications of Temporal Logic to the Specification of Real Time Systems," *Lecture Notes on Computer Science 331*, pp. 84–98, 1988.

80. C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, Vol. 36, No. 12, pp. 1425–1439, 1987.

81. W. Pugh, "An Improved Cache Replacement Strategy for Function Caching," *Proceedings of the ACM Conference on LISP and Functional Programming*, pp. 267–276, 1988.

82. W. Pugh and T. Teitelbaum, "Incremental Computation Via Function Caching," *Conference Record of the ACM SIGPLAN-SIGACT '89 Symposium on Principles of Programming Languages (POPL'89)*, pp. 315–328, 1989.

83. B. Ramkumar and L. Kale, "A Join Algorithm for Combining AND Parallel Solutions in AND/OR Parallel Systems," *International Journal of Parallel Programming*, Vol. 21, No. 1, pp. 67–107, 1992.

84. L. Rauchwerger and D. Padua, "The LRPDT test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pp. 218–232, 1995.

85. J. Snyder, D. Whalley and T. Baker, "Fast Context Switches: Compiler and Architectural Support for Preemptive Scheduling," *Journal of Microprocessors and Microsystems* (to appear).

86. M. Solal, D. Pillon and S. Brasseur, "Simultaneous Detection and Target Motion Analysis from Conventional Passive Beamforming Outputs," *Proceedings of the 1991 International Conference on Acoustics, Speech, and Signal Processing (ICASSP'91)*, Toronto, Canada, V2, 1991.

87. M. Spezialetti and R. Gupta, "Timed Perturbation Analysis: An Approach for Non-Intrusive Monitoring of Real time Computations," *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

88. G. L. Steele Jr., et. al., "Fortran at Ten Gigaflops: the Connection Machine Convolution Compiler," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI'91)*, pp. 145–156, 1991.

89. A. Stoyenko, "A Schedulability Analyzer for Real-Time Euclid," *Proceedings of the IEEE 1987 Real-Time Systems Symposium*, pp. 218–225, December 1987.

90. A. Stoyenko, "A Real-Time Language with A Schedulability Analyzer," Ph.D. Thesis, Department of Computer Science, University of Toronto, Canada, 1987.

91. A. Stoyenko, *Language-Independent Schedulability Analysis of Real-Time Systems.* in Real-Time Systems Engineering and Applications, edited by S. Pferrer and M. Schiebe, Kluwer Academic Publishers, Boston, 1992.

92. A. Stoyenko, "Evolution and State-of-the-Art of Real-Time Languages," *Journal of Systems and Software*, Vol. 18, pp. 61–84, April 1992. (also in Tutorial on Specification of Time — Abstractions, Design Methods and Languages, K. M. Kavi (Ed.), Washington: IEEE Computer Society Press, 1992.)

93. A. Stoyenko, "SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls," *Software-Practice and Experience*, Vol. 24, No. 1, pp. 27–49, January 1994.

94. A. Stoyenko and T. Baker, "Real-Time Schedulability-Analyzable Mechanisms in Ada 9X," *Proceedings of the IEEE*, Special Issue on Real-Time Systems, pp. 95–107, January 1994.

95. A. Stoyenko, V. Hamacher and R. Holt, "Analyzing Hard-Real-Time Programs for Guaranteed Schedulability," *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, pp. 737–750, August 1991.

96. A. Stoyenko and T. Marlowe, "Schedulability, Program Transformations and Real-Time Programming," *Proceedings of the IEEE/IFAC Real-Time Operating Systems Workshop*, Atlanta, Georgia, May 1991.

97. A. Stoyenko and T. Marlowe, "Polynomial-Time Transformations and Schedulability Analysis of Parallel Real-Time Programs with Restricted Resource Contention," *Journal of Real-Time Systems*, Vol 4, No. 4, pp. 307–329, November 1992.

98. A. Stoyenko, T. Marlowe, W. Halang and M. Younis, "Enabling Efficient Schedulability Analysis through Conditional Linking and Program Transformations," *Control Engineering Practice*, Vol 1, No. 1, pp. 85–105, January 1993.

99. A. Stoyenko, T. Marlowe and M. Younis, "A Language for Complex Real-Time Systems," *The Computer Journal*, Vol. 38, No. 4, pp. 319–338, November 1995.

100. A. Stoyenko, L. Welch, and B. Cheng, "Response Time Prediction in Object-Based, Parallel Embedded Systems," *Microprocessing and Microprogramming*, Vol. 40, pp. 135–150, 1994.

101. W. Strack, *Implementations of Distributed PROLOG*. Series in Parallel Computing, Wiley, New York, 1992.

102. R. Strom, et al., *Hermes: a Language for Distributed Processes*. Addison-Wesley, Reading, Massachussetts, 1991.

103. R. E. Strom and S. A. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 3, pp. 204–226, August 1985.

104. G. Tsai and B. McMillin, "Formal Methods of Real-Time Systems," *Technical Report Number CSC 91-17*, Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri, 1991.

105. J. D. Ullman, *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, 1982.

106. G.A. Venkatesh, *A framework for construction and evaluation of high-level specifications for program analysis techniques.* Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI '89), pp. 1–12, Portland, Oregon, June, 1989.

107. A. Vrchoticky, "Compilation Support for Fine-Grained Execution Time Analysis," *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

108. H. F. Wedde, B. Korel and D. M. Huizinga, "Static Analysis of Timing Properties for Distributed Real-Time Programs," *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, pp. 88–95, May 1991.

109. D. Whitfield, and M. L. Soffa, *Automatic Generation of Global Optimizers.* Proceedings of The ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91), pp. 120–129, Toronto, Canada, June 1991.

110. M. Wolfe, *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, Massachussetts, 1989.

111. Y. Wu and T. Lewis, "Parallelizing While Loops," *Proceedings of International Conference on Parallel Processing*, Vol. 2, pp. 1–8, 1990.

112. H. Yamana, et al., "A Macrotask-level Unlimited Speculative Execution on Multiprocessors," *Proceedings of the International Conference on Supercomputing (ICS'95)*, Barcelona, Spain, pp. 328-337, July 1995.

113. M. Younis, T. Marlowe and A. Stoyenko, "Compiler Transformations for Speculative Execution in a Real-Time System," *Proceedings of the* 15^{th} *Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

114. M. Younis, T. Marlowe, G. Tsai, A. Stoyenko, "Applying Compiler Optimization in Distributed Real-Time Systems", *Proceedings of the* 2^{nd} *IEEE International Conference on Engineering of Complex Computer Systems ICECCS'96*, Montreal, Canada, October 1996 (to appear).

115. M. Younis, P. Sinha, T. Marlowe and A. Stoyenko, "Performance Enhancement of Various Real-Time Image Processing Techniques Via Speculative Execution", *Proceedings of the IS&T/SPIE Symposium on Electronic Imaging: Science and Technology*, San Jose, California, Jan 1996.

116. M. Younis, G. Tsai, T. Marlowe, and A. Stoyenko, "Formal Verification of Transformation Rules for Speculative Execution in a Real-Time System," *Proceedings of the IFIP/IFAC Workshop on Real-Time Programming WRTP'95*, Ft. Lauderdale, Florida, November 1995.

117. M. Younis, G. Tsai, T. Marlowe, and A. Stoyenko, "Improving the performance of Fault Tolerance in Real-Time Systems Using Speculative Execution," *Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems ICECCS'95*, Ft. Lauderdale, Florida, November 1995.