Statically Safe Speculative Execution for Real-Time Systems

Mohamed F. Younis, Member, IEEE, Thomas J. Marlowe, Member, IEEE Computer Society, Alexander D. Stoyen, Senior Member, IEEE, and Grace Tsai, Member, IEEE

Abstract—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, we show 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 paper proposes a set of compiler transformation rules to identify opportunities for speculative execution and transform the code. Moreover, we have conducted an extensive experiment using simulation of randomly generated real-time programs 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 have been evaluated for realistic applications.

Index Terms—Real-time systems, speculative execution, shadow execution, compiler transformations, static analysis, distributed computation.

1 INTRODUCTION

In real-time applications, correctness cannot be separated from response time. These applications must provide outputs without violating the timing constraints imposed by the nature of the applications. Some applications have hard deadlines which, if missed, may result in disasters. For example, a late image identification of an enemy aircraft can cost the lives of many souls. Rapid image identification, remote command and control, avionics, and medical applications fall into this category. Other applications may have soft deadlines, such as broadcast and multimedia communication and high speed modeling. Ensuring timeliness while achieving satisfactory performance is a principal goal in study and implementation of real-time systems. When an application as written is not timely, code must be modified by automatic or semiautomatic means if possible.

Timing behavior of both soft and hard real-time systems must be predictable. Schedulability analysis [25], [34], [46], [48] refers to the compile-time prediction and verification of whether a program will satisfy its timing constraints. Schedulability analysis relies on predicting worst-case execution times (WCETs), which may depend on variable contention or synchronization times. In schedulability analysis it is assumed that for a given runtime instance of a statement, once the instance has been executed, its execution will never need to be repeated or undone.

In contrast, speculative execution refers to executing code without certainty whether the execution will be committed. Typically, speculative (or optimistic) execution [22], [53] 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 necessary. Strategies for speculative execution involve maintaining virtual time [22], [27], [53] and performing data replication [15].

These two techniques seem to be at the opposite ends of a continuum of time-management strategies for real-time systems. Consider a linearized execution of the code of the set of processes in a real-time system, where only synchronizing statements run in parallel. For each conditional encountered in the execution trace, at most one of its branches will be executed. Speculative execution may:

- execute a statement with outdated values, and need to retract the computation and reexecute it with the correct values.
- execute one branch of a conditional, and then need to retract that computation and execute a different branch, or none at all.
- make unnecessary calls or calls with invalid parameters, which will need to be retracted, if they have been executed, or killed, if they have not.

1.1 Speculative Execution in Real-Time Systems

Easy examples exist to show that, even when speculative executionprovably improves expected performance, it may result in failure of schedulability analysis and missed deadlines. In Fig. 1, assume exp involves a call and takes time 8,
Program transformations can be used to improve the timeliness, performance, and analyzability of real-time programs. However, to employ such transformations, one must show they are correct (both semantics and temporally), profitable, and automatable. We have developed a set of compiler transformation rules which, given a timing analyzer and dependence information as input, facilitates the use of speculative execution in complex real-time applications. The rules preserve not only program semantics but also timeliness [63], and can be incorporated into a real-time language compiler to be systematically applied. While applying these rules increases compilation overhead for real-time program, we show that speculative execution pays off.

Note that for real-time programs there are at least two possible definitions of temporal safety. In our approach, and in the related approach of [30], we require the elapsed time to any observable event not to increase; that is, there is no increase in WCET for any program fragment ending at an event. We will also require continued satisfaction of absolute and relative minimum constraints, which however we do not consider further here. This criterion is independent of deadlines, which will be met in the transformed program if they are met in the original. This approach can be applied even if precise deadlines are not yet known, or for reusable code.

We can also use information on actual deadline and waiting times to allow additional transformations which may slightly increase WCET while provably not violating timing constraints. In addition, such information can also be used to focus the efforts of the transformation system on regions where timing constraints are particularly tight. If the schedulability criterion is violated, and there are spare processors, we can view speculative execution as forking off an additional process, presumably lowering the load per processor, and (perhaps) enabling the system to be scheduled. In addition, speculative execution can improve other properties of real-time systems, such as fault tolerance [64].

1.2 Why Speculative Execution?
There are obviously several different approaches to improving the quality of a real-time application, each applicable to particular domains, application characteristics, and coding styles. We can identify the following classes:

- **Performance-driven optimization.** It is achieved using standard compiler optimization within tasks, and typically within procedures, guarded for temporal safety. The transformation either reaps the time gained, or replaces it by delays for later transformations [62]. This approach is suitable for compute-intensive procedures, automatically inlined or macro-expanded source, specializable library calls, or applicability of partial evaluation [36].

- **Constraint-driven optimization.** It is generally conducted through transformations which preserve (or even slightly worsen) performance, but are guaranteed to improve constraint satisfaction. Examples of these transformations include shifting nonreal-time code out of tight event blocks [13] and shifting an event with an absolute constraint above one with a relative constraint. Constraint-driven optimization are of most benefit when tasks (and procedures) contain either real-time and nonreal-time elements, or multiple, largely independent events.

- **Data parallelization.** It is usually implemented by applying standard loop parallelization and related techniques [59]. Data parallelization is most appropriate for compute-intensive tasks within a noncommunication-bound environment such as numerical, image processing and graphics, and multimedia applications.

- **Control flow parallelization.** It uses a collapsed data flow graph of the program to identify subtasks inside current tasks, and fork into separate threads of control [59].
Speculative execution. It is very suitable when there is significant conditional flow with tight constraints, and the application is not overly communication-bound [60].

Communication optimizations. These include elimination of redundant messages or parameters, separation and placement of calls and returns between modules, and optimization of placement and content of synchronous and asynchronous communication between tasks [40, p. 96]. This approach is appropriate for communication-intensive tasks with constraints crossing communication events, and for long chains of messages and/or calls.

Resource access transformations. They are particularly relevant to real-time systems. This approach improves the schedule of accesses to devices, shared memory, and other resources. They are most useful when there are significant shared resources, and when the resource access or scheduling disciplines are simple and subject to analysis [17], [49], [62].

For particular classes of applications, several of these may be simultaneously applicable, and the proper phase ordering for transformations is, even for nonreal-time applications, still very much open to investigation [58]. In particular, in classes for which speculative execution is applicable, standard optimizations, one or both forms of parallelization, and resource access and communication optimizations may also lead to better performance and satisfaction of temporal constraints. The issue of which approaches are preferable, singly or in combination, or the order in which to attempt transformations, is very much a matter for future study.

Our work in particular has focused on speculative execution for three reasons. First, it is the least well explored of all of the intraprocedural optimization techniques. Second, by comparison to parallelization, we are convinced that large classes of real-time and process control applications do have conditionally controlled and temporally constrained events, and constraints may be too tight to afford stopping after the condition is evaluated to fork off a process. Third, in comparison to optimization, there have already been some preliminary studies of optimization for real-time processes [13], [30]; the first of these is constraint-directed and fairly straightforward, while the second would entail deriving, specifying, and proving precise real-time versions of a suite of optimizations, since a single optimization is unlikely to lead to significant improvement by itself.

In the future, we hope to extend the simulation, presented in Section 6, to recognize opportunities for safe parallelization, and for optimization of communication, and to compare their effectiveness with techniques for speculative execution, both head-to-head and in various phase orderings.

We can also identify specific and significant real-time image processing applications which can benefit from speculative execution. Image filtration, for example, usually involves a lot of computation, while testing the quality of an image is time-consuming as well [9]. An image can be filtered speculatively on a shadow while quality tests are running on the primary. The same argument holds for edge detection. Moreover, morphological image processing [14] has a lot of potential for speculative execution. Construction of a structural element can be performed 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 WCET. Similar examples from real-time artificial intelligence, autonomous vehicle control, and other applications could likewise be provided.

This paper is organized as follows. We present related work in Section 2. Next, we define the computation model and identify opportunities for speculative execution in Section 3. We illustrate our transformations by an example. Section 4 elaborates on various safety issues affecting the applicability of speculative execution to real-time programs. The specification of compiler transformation rules are provided in Section 5, including an overview of a formal semantic correctness and timeliness proofs. Section 6 illustrates the design and results of an experiment based on simulation. In Section 7, a prototype implementation of the transformation rules is described and the applicability of the speculative execution transformations to real-life applications is discussed. Finally, Section 8 concludes this paper.

2 RELATED WORK

Speculative execution has been addressed on the machine instruction level in superscalar and VLIW machines such as [3]. However, real-time issues have not been addressed. In this paper, we do not look at that level of granularity, but extract opportunities at source code level. Rollback has been used in distributed environments for other purposes such as synchronizing processes [22], and performing recovery from failure [53]. A semistatic approach to speculative execution is presented in [26]. The approach uses profiling to collect information about the correlation between branch conditions and replicates the code of the most probable branches. In addition, speculative execution is applied to database management in [7], [54].

Compile-time analysis of real-time systems has been used to predict timing behavior of real-time programs by annotation [31], [39], [57], partial evaluation [35], [36], or through simulating architecture specific behavior [29], [32], [33]. In addition, compiler transformations have been used to schedulability analysis [49], [50], to adapt monitoring activities [45], and to perform code optimization [30], [56]. Moreover, compile-time analysis has been used to enhance the system schedulability through safe code motion [13], [20], [16], extracting potential for execution overlapping [17], allow efficient context switching by remapping registers [43]. We mainly apply speculative execution transformations to enhance the average performance, but system schedulability can be improved as well.

3 OPPORTUNITIES FOR SPECULATIVE EXECUTION

Recall that speculative execution occurs when we execute code without being certain as to whether or not the execution will be committed. In this section, we identify opportunities for speculative execution and illustrate our approach to analyzing applicability and profitability of
speculative execution. We begin with the problem model assumed throughout the paper.

3.1 A Problem Model

We assume a set of periodic top-level processes, 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.

Our analysis relies on an expressive real-time language with 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 with more general loops and limited recursion [10], [51]. However, we assume the language allows concurrency and interprocess synchronization. It is also assumed that the execution time of the program can be predicted. A machine-dependent timing analysis is to be performed to handle modern processor architectures with advanced features, such as cache memory and instruction pipeline. Moreover, there should be an upper bound on communication delays. Although, the longest execution path of a real-time program might actually take less than the predicted worst-execution time, the predicted worst-case execution time, even if it is pessimistic, has to be considered in the system schedulability analysis [13], [25], [31], [35].

Throughout the paper we use the following data dependence terminology describing dependences between the code segment $S$ and another code segment $P$:

- **True** or flow dependence. Value of a variable $x$ set in $S$ reaches use in $P$.
- **Anti**-dependence. Value of a variable $x$ used in $S$ may subsequently be set by a definition in $P$.
- **Output** dependence. Value of a variable $x$ set in $S$ may be overwritten by the definition in $P$.
- **Input** dependence. Value of a variable $x$ used in $S$ may 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 $P$ while resource $R$ is ordered (need not be shared with other processes).
- **Data** dependence. true, anti, and output dependence (input dependence typically matter only in the presence of memory hierarchies).

In the following sections, we discuss possible opportunities for speculatively executing branches of conditionals and while loops.

3.2 Opportunities for Speculatively Executing Conditionals

Assume that we have a call at a branch point (for simplicity we assume exactly two branches) and the code is of the form $s$: if ($C$) then $S_2$ else $S_3$, where $C$ is a call being executed on another processor. If we store (and possibly later retrieve) the current state at $s$, there will be a cost a time delay of $t_c$ for the copying (store), and $t_r$ for the restore.

If the execution time of $S_2(Time(S_2))$ dominates that of $S_3(Time(S_3))$, and $Time(S_2) - Time(S_3) > t_c$, and further, some initial segment of $S_2$ is not data dependent on an output parameter of $C$, then we can begin executing $S_2$ speculatively, abandoning the computation and restoring prior state only if the returned value indicates we should have chosen $S_3$. 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 $C$ (or 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 restore is inexpensive, and that branch has some initial segment not data dependent on $C$, we can speculatively execute that branch. (If both branches have this property, we use the one with the longer execution time.) Furthermore, if there are some data dependencies on a value modified in $C$, we can speculatively execute that branch and stop at the point when we use that value, provided that this is profitable.

3.3 Opportunities of Speculatively Executing While Loops

While the model of [46] 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 [24], [42], [59]. For while loops, this may involve speculative execution of some number of iterations, saving the state after each. In speculative execution, the next loop iteration may be evaluated in parallel with a call made near the end of the previous iteration, where the loop condition depends on the return value [23] (or in our case, a call inside the condition itself). For example, in the code block $s$: 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 condition evaluation 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, we can remember the original values, allow iteration to proceed, and restore precisely the values which have been overwritten by speculatively executed iterations which do not in fact occur.

Again, attention should be paid to the worst-case scenario. The costs of rollback must be estimated, and transformations not performed, if these endanger satisfaction of timing constraints.

3.4 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 frequently used in databases [7], [8], [54]. That
is, checkpoint-and-restore (which we call “speculative execution” hereafter) 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 modiﬁng 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 we can improve both expected and worst-case performance.

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, we may even be able 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, we can speculatively evaluate both branches, each on a different processor. Results of both branches are transmitted, but one will be discarded once the call returns.

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

Note that for real-time programs, both classical speculative execution and shadow execution must execute the longest arm of a conditional. In general, for nonreal-time programs, the choice of arm and of technique may depend on estimated branch probabilities; for real-time programs, branch probabilities can only be used for proﬁtability estimates.

### 3.5 An Example of Code Transformation

Considering the code fragment of Fig. 2, suppose that none of stmt_1 through stmt_k uses the parameters x and y of the method call m(x, y). We can then speculatively execute these statements 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. We call the block containing m 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.)

![Fig. 2. Code transformation for speculative execution.](attachment:image.png)

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 our 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 came 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. In such a situation, the speculative call should not be made. The program call graph can be consulted to safely detect such a case.

Note that the fork construct can be generalized to an arbitrary multway 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.

### 4 Issues of Speculative Execution for Real-Time Systems

In this section, we address issues related to applying speculative execution to real-time systems. We begin with issues of safety concerning timeliness and data ﬂow dependence, and then consider possible interaction with real-time optimization techniques.
4.1 Ensuring Timeliness
Speculative execution can be considered as a program transformation which enhances concurrency of program's execution. The important issue is whether the transformation for speculative execution ensures, or more importantly, preserves timeliness. It is possible, although not guaranteed, that transformations for speculative execution may even improve a program's deadline satisfaction; however, a poorly chosen transform can make it difficult to satisfy deadlines. Therefore, it is crucial to investigate the effect of the transformation on the worst-case timing behavior of the program. Safety checks must of course consider the overhead of forking new processes and committing or aborting the results of speculative execution, and the effect of rollback has to be bounded. Any transformation must not only preserve or improve worst-case execution time (WCET); the transformation also must be profitable, enhancing worst-case or average execution time.

In order to perform such safety and profitability analysis, the execution times of various parts of the program need to be predicted a priori. Although bounding the execution time of real-time programs is a challenge, especially for advanced architectures with cache memory and instruction pipelines, many techniques, for example [29], [32], [33], have been proposed in the literature for such architectures. We assume that a timing analysis is available as input to our transformation rules.

Note, however, that in the presence of such architectural features, a transformation may change instruction sequence or cache behavior, and thus WCET; therefore, it may be necessary to perform a machine-dependent timing analysis on a portion of the transformed program containing the changed region. This machine-dependent analysis may further tighten the bound on the execution time for the parts of the code subject to transformation, and consequently enhance the precision of the safety and profitability analysis of the transformations. The interaction between the speculative execution transformer and the timing analysis tool is not further addressed in this paper.

It must be noted that WCET computations are done in two stages: basic times in the absence of contention and times which include contention. The former computation depends on assumptions about the predictability of the programming language implementation, hardware and so on. The accuracy of this prediction will thus be essentially independent of the schedulability method used but will reduce to the accuracy of the implementation assumptions. The latter computation causes an exponential growth in the number of possibilities to consider for accurate prediction. Since we cannot consider all these possibilities (the problem is NP-hard), we employ other transformations of conditional [49], which are the main source of this exponential growth, as well as conditional linking [50], [52]. Naturally then, symbolic execution assumptions must be used, as per programmer's best guess. Should the programmer guess inaccurately, some infeasible combinations of conditional selections may be assumed feasible. Since the WCET is the maximum over all considered possibilities, consequently, any WCET prediction of the latter kind may include an overly pessimistic estimate, should these infeasible paths be considered feasible for the purpose of the analysis. Observe, however, that the problem will not be with the analysis (ours or anyone else's) but rather with the incorrect symbolic execution assumptions. Thus, independently of the schedulability method chosen, the possibility exists that WCET may be predicted somewhat inaccurately. In the balance of the discussion, we will thus mean "predicted WCET" when we say "WCET."

4.2 Ensuring Correct Semantics
Timeliness is not the only property to be preserved during transformation; we must of course consider data dependence as well. The immediate issues are, first, that we may need in speculatively executed code to use a value still being computed in parallel (true or input dependence), or, second, that other variables may get the wrong values, or at the wrong time, or in the wrong order (anti or output dependences). While the second of these is not a problem because of deferred commitment, the first will generally inhibit some form of speculative execution. However, we may be able to use current values in the speculative computation if it is possible to quickly adjust the results of the speculative execution before committing them.

Other constructs may also inhibit speculative execution. First, 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). We cannot, in general, afford to checkpoint memory reached from an arbitrary live variable. However, we can use shadow memory and dispose only of the copy until the speculative execution commits. For example, if a pointer is to be dereferenced, we can create a shadow copy pointing to the same structure and free the original pointer. If the execution is committed, we also free the shadow copy. Otherwise, we restore the pointer from the shadow.

Second, speculative writing to output or reading from input may cause inconsistency. Writing can often be handled by intercepting the writes in a buffer. Although writing on commit may produce invalid semantics if multiple processes can access the buffer, all subsequent writes could be buffered until the speculatively executed code aborts or commits. The same approach can be used for input by buffering the values read from input and providing them to their eventual targets.

Finally, 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. More importantly, some exceptions have persistent effects on the environment; these effects have to be intercepted and buffered.

4.3 Changes in WCET: Blocking and Architectural Effects
In our presentation of various opportunities for speculative execution, we have assumed that the WCET of the speculatively executed code S (excluding fork/join and save/restore) is identical to that for the code if executed in place. There are however several possible complications. The most obvious concern is blocking; can the speculatively executed code experience different blocking from the original?
Blocking can occur for resources, objects, or communication, and either between 
S and the original task T, or between S and the other tasks in the system. To avoid blocking 
between S and T, we can check that there are no common (or no conflicting) accesses to resources or objects, and that the speculative code does not commit communication to other tasks, or to the external environment. We also check (or are guaranteed by the architecture) that the fork and join messages do not interfere with other messages.

However, requests from S to resources or objects may collide with requests from tasks other than T. It is even possible that for some scheduling disciplines moving the request from S earlier in time may not affect S but still change the order in which results from other tasks are served. For this latter problem, we can assume that transformation occurs after static scheduling, and does not in this phase affect the service time for other processes’ requests, except by direct collision. For the former situation, we have three possible approaches:

1) to identify and disallow any such situations;
2) since transformations are local (although possibly nested or interacting), to conditionally commit the transformation and later perform a blocking analysis; and
3) to analyze blocking on a case-by-case basis, possibly necessitating a reanalysis of WCET for S.

Even in this case, we would not expect substantial problems: since the global WCET analysis for the program treats inter-task conditions as independent, the current analysis already allows for the requests generated by S to be handled when S is executed in place. We then need only to show that earlier arrival of those requests does not delay completion time of other requests, and that S’s requests complete early enough for the safety and profitability of the transformation. We do not consider this issue further in this paper.

There is another effect to be considered with modern architectures: the cache, pipelining, instruction scheduling and other effects might change, since S is typically being executed in isolation in a virgin environment. For such architectures, the WCET of S may need to be recomputed, although an incremental recomputation may suffice.

4.4 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 $y := a_0x^n + a_1$, with data dependence on the value of a variable x which is computed in an earlier step as $x := c1$, where c1 is time-consuming call. We may assume a value $c_0$ for c1 and later update the value of y affected by that assumption. When the true value c1 is returned, we can use c1 - $c_0$ as an increment to x, and adjust y using the finite differencing technique described in [37], [38].

Thus, the computation of y can be performed during the call c1, and can be adjusted upon the conclusion of the call. The code added to make that adjustment is called "δ-code." Generally, if the dependence on the return value is "simple," so that we can easily specify δ-code, and the time to return from the callee is greater than the time for the δ-code update, we can speculatively evaluate the code 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, we can cache parameter patterns and return values, and reuse return values instead of making a call (providing that testing of equality for parameter lists is cheap) [41]. If tests are not cheap, but we can pass kill messages, we may start a call, and store initial state at the callee. If we later find that the call is unnecessary, we send a kill message. In some cases, we may want 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 [28]. These techniques are not explored further in this paper.

Since the composition of improving transforms will be improving, the technique will also interact with more standard compiler transformations for optimization and parallelization [2], [30], [59] as specialized for real-time programs. Nonetheless, one improving transformation may inhibit another, and the appropriate sets of transformations, orderings, and strategies are a matter for future exploration.

5 Transformation Rules for Speculative Execution

In this section, we developed a set of compiler transformation rules for speculative execution [61], as discussed in Section 3. We begin with our notation, followed by specification of the transformation rules.

5.1 Representation of the Rules

We use an axiomatic specification approach 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 [55], [58], for specifying data dependence and control flow conditions. However, they are not in their present form suitable for real-time systems, since compiler transformations of real-time programs cannot ignore timing constraints and resources access. The rules are specified by standard Hoare triples [19]:

(precondition, action, postcondition)

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

1. This is related to our approach to multiprocess optimization [62]. A similar global analysis will also need to realize improvements made possible in other tasks by transforming the current task.
• 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. We assume that resource dependences have been captured in Blocking or Ordered constraints on resource access (see next). We also assume that effects of the transformation on resource queues and access by other tasks are resolved as discussed previously.

• 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.

We use the following information 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 [2].

• 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; 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, we assume the following timing functions:

1) The variable used in S₂ are not modified as a side effect of the call C.

2) Neither S₂ nor any function transitively called from S₂ has a blocking construct.

3) There will not be a change in the order of any calls to critical sections if S₂ runs while C is running.

To ensure the safety and profitability of the transformations, timing conditions of (7) and (8) in Fig. 3 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 (storage, forking, and joining) is less than the execution time of C, which is also guaranteed by (7).

It should be noted that in case of rollback the shortest branch may take longer to execute compared to the time required in the absence of speculative execution. This may introduce some slowdown in the nonworst-case execution time. However, most of the time (if not all) the code block S will be executed in less than the predicted worst-case time and thus enhancing the average performance and making the real-time program more robust. The timing preconditions ensure the timeliness of the transformed program, which is an essential property for real-time systems, by verifying that the shortest branch will not be extended over the predicted nonspeculative worst-case execution time.

The Speculative-While rule, in Fig. 4, follows in the same spirit. The shadow execution rules, Fig. 5 and Fig 6, are based on the same preconditions considering copy-and-commit rather than rollback.

5.3 Formal Verification of the Transformation Rules

In the previous subsection, we presented the compiler transformation rules for speculative execution. The rules must preserve program semantics and prevent extending the worst-case execution time. In this subsection, we provide an overview of the formal verification of semantic correctness and timeliness for speculative execution transformation rules. For a detailed discussion and proofs, the reader is referred to [60], [63].

A correctness proof of a (nonreal-time) compiler transformation consists of three parts: 1) showing that the data flow equations abstract the program semantics; 2) proving that the data flow computation terminates; and 3) proving that the transformation preserves 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.

It has been known that the formal verification techniques developed for shared-memory multiprocessor apply equally well to distributed systems and real-time systems (with an additional variable time) [1], so the temporal logics of concurrent systems [4], [6], [18] can be applied to real-time
Fig. 4. Speculative execution for \texttt{if} clauses.

**RULE:** \hspace{1cm} \texttt{SPECULATIVE\_IF}  
**Preconditions:** Structural:  
(1) $S = (\text{if } (C) \text{ then } S_2 \text{ else } S_3)$ is a single-exit code region.  
(2) $C$ is a call being executed on another processor  
Dependence:  
(3) $\text{Vars}(S_2) \cap \text{Mod}(C) = \emptyset$  
($S_2$'s variables have correct values immediately before \texttt{if})  
Blocking:  
(4) There are no blocking constructs in $S_2$.  
(5) For all methods $M$ in $\text{TCalls}(\text{Calls}(S_2))$, not $(\text{Blocking}(M))$.  
(6) For each method $M$ in $\text{TCalls}(C) \cap \text{TCalls}(\text{Calls}(S_2))$, not $(\text{Ordered}(M))$.  
\hspace{1cm} \text{(Incorrectly or prematurely executing any such statement has a permanent and invalid effect on the environment.)}  
Timing:  
(7) $t_f(\text{Mod}(S_2)) + t_f + t_f(\text{Mod}(S_2)) + \text{Time}(S_2) \leq \text{Time}(C)$.  
\hspace{1cm} \text{(Useful work can be done; worst-case time does not increase.)}  
(8) $\text{Time}(S_2) + t_f(\text{Mod}(S_2)) \leq \text{Time}(S_2)$.  
\hspace{1cm} \text{(Worst-case time does not increase.)}  
Actions:  
\hspace{1cm} \text{Execute } C \text{ in parallel with the following:}  
\hspace{1cm} \text{save(\text{Mod}(S_2)); } S_2.  
\hspace{1cm} \text{Insert synchronization between } \text{exit}(C) \text{ and } \text{exit}(S_2).  
\hspace{1cm} \text{Check } x_v, \text{ the return parameters of } C;  
\hspace{1cm} \text{If this enables } S_2, \text{ do nothing.  }  
\hspace{1.5cm} \text{Otherwise, } \text{execute } \text{restore(\text{Mod}(S_2)); } S_3.  
\hspace{1cm} \text{In any case, continue executing from } \text{exit}(S).  
\hspace{1cm} \text{Postcondition: } S \text{ has completed without missing its deadline.}  
\hspace{1cm} \text{State is as if execution had been sequential.}  
\hspace{1cm} \text{Comment: } A \text{ symmetric rule exists for } S_3.  
\hspace{1cm} \text{Properties: } \text{Preserving the program semantics.}  
\hspace{1cm} \text{Not extending the worst-case execution path.}

Fig. 3. Speculative execution for \texttt{while} clauses.

**RULE:** \hspace{1cm} \texttt{SPECULATIVE\_WHILE}  
**Preconditions:** Structural:  
(1) $S = (\text{while } (C) \text{ do } S_2)$ 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) $\text{Vars}(S_2) \cap \text{Mod}(C) = \emptyset$  
($S_2$'s variables have correct values immediately before \texttt{while})  
Blocking:  
(5) There are no blocking constructs in $S_2$.  
(6) For all methods $M$ in $\text{TCalls}(\text{Calls}(S_2))$, not $(\text{Blocking}(M))$.  
(7) For each method $M$ in $\text{TCalls}(C) \cap \text{TCalls}(\text{Calls}(S_2))$, not $(\text{Ordered}(M))$.  
Timing:  
(8) $t_f(\text{Mod}(S_2)) + 2t_f(\text{Mod}(S_2)) + 2t_f + 2t_f + \text{Time}(S_2) \leq 2\text{Time}(C)$.  
\hspace{1cm} \text{(Useful work can be done; worst-case time does not increase.)}  
(9) $t_f(\text{Mod}(S_2)) \leq \text{Time}(S_2)$.  
\hspace{1cm} \text{(Given at least one iteration; worst-case time does not increase.)}  
Actions:  
\hspace{1cm} \text{Execute } C \text{ in parallel with the following:}  
\hspace{1cm} \text{save(\text{Mod}(S_2)); } S_2.  
\hspace{1cm} \text{Insert synchronization between } \text{exit}(C) \text{ and } \text{exit}(S_2).  
\hspace{1cm} \text{Check } x_v, \text{ the return parameters of } C;  
\hspace{1cm} \text{If this enables } S_2, \text{ repeat.  }  
\hspace{1.5cm} \text{Otherwise, } \text{execute } \text{restore(\text{Mod}(S_2)); } \text{exit}(S).  
\hspace{1cm} \text{Postcondition: } S \text{ has completed without missing its deadline.}  
\hspace{1cm} \text{State is as if execution had been sequential.}  
\hspace{1cm} \text{Properties: } \text{Preserving the program semantics.}  
\hspace{1cm} \text{Not extending the worst-case execution path.}
programs. Temporal logic allows the abstraction and validation of a property of a program without knowing all the other properties. Thus, temporal logic has been adapted since it is important to prove that if a property, in particular program semantics or timeliness, holds originally in a real-time program, it will hold after applying the transformation rules.

**Semantic Correctness Proof.** The approach to proving that the transformation rules preserve program semantics is to use temporal logic, enhanced with a denotational-semantics-like representation of program stores. Let $\Sigma$ denote the set of states of a program $P$ and let $\sigma_s$ denote the state of a program after executing the code segment $S$ where $\sigma_s \in \Sigma$. Let $\sigma(x)$ be the value of a variable $x$ at the state $\sigma$. Given $\sigma_s$ and $\sigma_s'$, we say that the states $\sigma_s$ and $\sigma_s'$ are equivalent if for every $x \in \sigma_s', x \in \sigma_s'$ and the value of $x$ is the same in $\sigma_s$ and $\sigma_s'$. Thus, the goal is to prove that applying a transformation rule should lead to a semantically equivalent state.

Considering the example in Fig. 1, assume that $\sigma_{\text{code block}1}$ and $\sigma_{\text{code block}2}$ denote the states, with no speculative execution, after the execution of $\text{code block}1$ and $\text{code block}2$, respectively. Suppose after applying the transformation rules $\sigma'_{\text{code block}1}$ and $\sigma'_{\text{code block}2}$ denote the states after the execution of $\text{code block}1$ and $\text{code block}2$, respectively. The transformation preserves the program semantics if

1) The states, $\sigma_{\text{code block}1}$ and $\sigma'_{\text{code block}1}$ are semantically equivalent, and

2) $\sigma_{\text{code block}2}$ and $\sigma'_{\text{code block}2}$ 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 Fig. 1, leads to a state which is semantically equivalent to the state without the extra computation.

The proof considers all live variables on exit of the transformed code block. Any temporary variable introduced by the transformation will not be referenced in other places in the code and therefore will not affect the semantics. In ad-
dation, it is proved that the transformed code terminates if the original code terminates. The detailed semantics correctness proof can be found in [60], [63].

Timeliness Proof. Timeliness should be guaranteed before the transformations are performed. Consider the example of Fig. 1. If the execution time of each block is as defined as in Section 1, 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 the timing proof, 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 by the speculative execution transformation. Using temporal logic, it has been shown that the state following the transformed code is reachable, in the worst-case, within time less than or equal to the time without the transformation. As above, collisions with other tasks are either prohibited or resolved. Again, the detailed proof can be found in [60], [63].

6 Experimental Results Using Simulated Workload

In previous sections, a set of transformation rules for speculative execution have been developed, and it has been shown that these rules neither change program semantics nor extend deadlines. There is still a question, nonetheless, of applicability, which is certainly not guaranteed: it may happen that rule preconditions are never met. Consequently, applicability cannot be validated using formal verification. In this section, various coding characteristics that affect the applicability of speculative execution in real-time programs are investigated using simulation. In the next, prototyping is used to address applicability and usefulness of speculative execution in (program fragments from) actual real-time applications.

Our investigations have shown that speculative execution can be effective for an industrial real-time application such as a cardiac workstation [50]. However, the applicability of speculative execution can potentially be affected by a number of code parameters, themselves dependent 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 decided to investigate the effects of coding parameters through simulation. The simulation uses randomly generated sets of real-time programs created using a workload generator. We have also looked at real-time programs for air traffic control [12], passive sonar [44], navigation control [13], multimotor control system [5], and quality monitoring [11]. The design and parameters of the simulation are largely guided by these applications.

Among factors affecting applicability of speculative execution to real-time software are data dependence, frequency of conditionals and while loops, and size of conditional clauses and while loop bodies. This section provides a study of the impact of such properties on the number of opportunities and program timeliness, as well as performance enhancement to be anticipated while applying speculative execution transformations.

The experiment is based on simulation using randomly generated programs, and consists of the following steps:

1) Generating Programs. A program is a group of statements. 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 [50], [12], [44], [5], [11], the following probabilities are assigned to each statement type: (IF, 10 percent), (LOOP, 10 percent), (ASSIGNMENT, 35 percent), (CALL, 20 percent), (BLOCKING_CALL, 5 percent), (READ/WRITE, 20 percent). We subsequently have explored the marginal effect of changes in each of these probabilities individually. We assume that READ and WRITE each use buffers, and can be considered nonblocking. A call is considered as an invocation of code on a different processor, and can be blocking (BLOCKING-CALL), or nonblocking (CALL). Parameters to calls are randomly selected from the set of variables and classified randomly as in or out parameters.

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. For while-loops, the same call will be included in the body of the loop, updating the condition variable. 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 to be used in computing the worst-case execution time.

Locality of reference entails some combination of lengths of live ranges and degree of reuse of variables. In the experiment, we take locality of reference to be a measure of the first of these, and have a separate “degree of reuse” parameter. To simulate locality of reference, a program is divided into segments, where segment size is controlled by a parameter determining percentage of locality (i.e., segment_size = program_size * (1-locality)). The variables of the program are partitioned among the segments (i.e., segment_variables = max_no_variables * (1-locality)); a random number specifies how many of these will be reused in the next segment. Two lists of variables are maintained for each statement: 1) the set of variables reached (used) in that statement; and 2) the set of all variables modified or defined. These lists are used while applying the speculative execution rules to ensure the dependency conditions.

2) Assigning Times to Primitive Statements. At this step, the worst-case execution time is attached to each statement. Execution time for primitive statements is assumed proportional to the number of variables referenced in that statement; consequently, the execution
Calculating the Average Execution Time

Applying Transformation Rules

Calculating WCET and Deadlines. The control-flow graph of the generated programs is analyzed and the worst-case execution time (WCET) is calculated. 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 WCET. In the experiment, the deadline is selected randomly in the range of \([0.8, 1.1] \times \text{WCET}\), reflecting a tightly constrained and likely overloaded task set. Thus, a mixture of processes can be provided, some of them meeting and others missing deadlines.

Calculating the Average Execution Time. The (non-speculative) average execution time \(T_{\text{TSE}}\) is affected by conditionals and the number of iterations of while loops. The probability of selecting the longest branch of an if is assumed to be 90 percent. For loops, a random number is 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.

Applying Transformation Rules. In this step, the compiler transformation rules are applied to the generated programs. Since, the generated programs have only explicit calls, there will be no call in the condition of if and while statements. We thus extended our analysis to consider cases in which the boolean expression of an if or while statement involves a variable modified by an earlier call which makes the granularity of the condition big enough to enable the transformations. 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 \(T_{\text{TSE}}\) \((T_{\text{shadow}})\), the algorithm of the previous step is used for serially executed code. However, in applying the transformation rules, the computation considers on the one hand parallel execution, and on the other overhead for task management. Forking and joining time, \(t_f\) and \(t_j\), include interprocess communication, including sending messages over links with 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. For 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 500 m/\(2 \times 10^8\) m/sec, which is 2.5 \(\mu\)sec; assuming at most 526 bytes for the variables, the transmission time is at most 526/10 Mbps, or 20 \(\mu\)sec [21]. 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 \(T_{\text{TSE}}\). 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 percent of the time.

Measuring the Impact of Speculative Execution.

Programs are classified into four groups depending on the values of \(T_{\text{TSE}}\) and \(T_{\text{TSE}}\) compared to the deadline. Programs for which \(T_{\text{TSE}}\) and \(T_{\text{TSE}}\) are greater than the deadline (Group 0) are of no interest since they miss deadlines anyway. Group 1 contains programs that always meet their deadlines. Programs that could meet their deadlines only with speculative execution is in Group 2. Since no code will be speculatively executed if it is not safe or profitable according to the analysis performed by the compiler, Group 3 is empty (i.e., programs cannot miss deadlines due to transformation), and we ignore Group 3 hereafter.

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 [0.6, 0.7] \(\times\) WCET than for the range of [0.8, 1.1] \(\times\) WCET. An experiment is conducted using 1,000 programs, assuming 5 percent to 10 percent if 's and loops. The result of this experiment is shown in Table 1.

Before transformation, programs in group 1 meet and programs in groups 0 and 2 miss their deadlines; afterward, those in groups 1 and 2 meet, and only group 0 misses its deadlines. Thus, the third row indicates that before applying speculative execution, there are 370 (630) programs meeting (missing) deadline, while (370 + 537 = 907) meeting deadlines after applying the transformation rules.

As indicated in Table 1, speculative execution may not help if deadlines \(D\) are too tight; for \(D \in [0.5, 0.7]\times WCET\), no programs meet their deadline, before or after. On the other hand, speculative execution will have a minor or no effect on timeliness if deadlines are slack, since almost all programs will already meet deadline, as for \(D \in [1.0, 1.1]\times WCET\).

<table>
<thead>
<tr>
<th>Deadline</th>
<th>Group 0</th>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.5, 0.7] (\times) WCET</td>
<td>1,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[0.7, 0.9] (\times) WCET</td>
<td>93</td>
<td>370</td>
<td>537</td>
<td>0</td>
</tr>
<tr>
<td>[0.9, 1.0] (\times) WCET</td>
<td>63</td>
<td>650</td>
<td>287</td>
<td>0</td>
</tr>
<tr>
<td>[1.0, 1.1] (\times) WCET</td>
<td>0</td>
<td>1,000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
6.1 Performance

In the simulation, the performance is measured by speedup, improvement of timeliness, and applicability, defined as follows:

- **Speedup**: the percentage of $1 - \frac{T_{SE}}{T_{NSE}}$.
- **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.

Each test set contains 1,000 simulated programs, each with 1,500 statements, selected according to the probability density function described in Section 6, with a set of 100 variables used in each program. Deadlines are in the range of [0.8, 1.1] * WCET. A 10 percent locality of reference with 50 percent reuse from the previous segment is assumed. To capture the impact of various parameters multiple experiments are performed changing one parameter at a time. In Figs. 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, and 19, SE and SSE are used to denote speculative execution and shadow execution, respectively. The following summarizes the results.

6.1.1 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. The simulation results confirm our expectation, as shown in Fig. 7. Fig. 8 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.

6.1.2 Size of Various Blocks

The effect of the size of the then-clause of a conditional on applicability and other performance measures Figs. 9 and 10. 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 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 Figs. 11 and 12.

6.1.3 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 Fig. 13). However, improvement in timeliness and speedup is getting saturated and even decreases after a certain threshold, as shown in Fig. 14. 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 Fig. 15), and on the speedup and timeliness (see Fig. 16).

As expected, with the increase of the percentage of blocking calls, the opportunities for transformation decrease (see Fig. 17). The greater chance of having blocking calls inside if and while statements results in difficulty of satisfying the blocking constraints.

6.1.4 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 nonlinear. 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 Figs. 18 and 19). 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.

7 Implementation and Test Environment

In the previous section, the impacts of various code characteristics on the applicability of speculative execution have been studied. While the results from the simulation 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 section includes a description of those efforts, testing applicability of the transformation rules of Section 4 in fragments of actual real-time applications.

3. Every program segment refers to 10 percent of the variables and half of those carry over from the previous segment.
Fig. 7. The relationship between opportunities and program size.

Fig. 8. The effect of program size on performance.

Fig. 9. Size of if blocks vs. opportunities.

Fig. 10. Impacts of the if block size on performance.

Fig. 11. Size of while blocks vs. opportunities.

Fig. 12. Impacts of the while block size on performance.
Fig. 13. If frequency vs. opportunities.

Fig. 14. Impacts of if frequency on performance.

Fig. 15. While frequency vs. opportunities.

Fig. 16. Effects of while frequency on performance.

Fig. 17. Percentages of blocking calls vs. opportunities.

Fig. 18. Locality of variables reference vs. opportunities.
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.

7.1.2 The Compiler

Inputs to the compiler include:

1) the source code,
2) a file of architectural specifications, including instruction-class/time maps, network topology, and other interconnection details, and
3) a file of compile-time assertions.

Compiler output will be intermediate code (in this implementation, in a safe subset of C++, including runtime checks, and a timing constraints file. The compiler also constructs standard control and data flow/dependence representations: a call graph, a control flow graph per process/method, and a data dependence representation, currently, use-def chains [2]. Correspondence between generated code and control flow graph is maintained by two pointers per basic block, to the start and end of the corresponding intermediate code segments.

7.1.3 The Timing Tool

The timing tool provides safe static estimates of execution times [46]. Inputs include the instruction timing map of the target architecture, given as a table of instruction type and required execution time. To resolve call execution time, the timing tool uses the call graph, unwound if necessary in the presence of (bounded) recursion. For remote calls, the tool assumes an upper bound on message propagation delay.

By browsing the control flow graph, the timing tool calculates two forms of execution times: first, the worst-case execution time (WCET) of each process, to resolve references to other methods through calls; second, a timing annotation on each executable statement, simple or structured, using the timing map. Execution time of each basic block is stored in the control flow graph. The WCET of the entire method/thread is then computed using basic-block execution time and times for called procedures, and stored with the method/thread entry in the call graph. Because only intermediate code is generated in the current implementation, a language-defined map is used for access to basic data types (classes) and for operations.

7.1.4 The Analysis and Transformation Engine

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

The transformation engine may apply multiple types of transformations, such as those discussed in the Section 2. The engine may use a sequence of steps, where in each step a different kind of transformation will be considered; it may however be desirable to repeat a step because of additional

---

Fig. 19. Effects of locality of variables reference on performance.
opportunities potentially introduced by other steps. For example, branch/clause transformations can be usefully reapplied if a condition can be eliminated by the conditional linker [50]. This dependence is represented by the feedback arrow (1) in Fig. 20.

7.1.5 The Schedulability Analyzer
The transformed code and the constraint file are passed to a schedulability analyzer. The schedulability analyzer may use either an exhaustive or a heuristic analysis to produce an assignment and a certificate of schedulability; the analyzer may also report a partial static schedule to be used by the runtime environment.

7.1.6 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 runtime environment, which simulates distributed processing of the code over a network of processors.

7.1.7 The Runtime Environment
The runtime 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.

The Kernel polls a priority-queue event list. Events include: scheduling a method/thread, executing a call to a method, sending a message to a remote object (that is, an object on a different processor), and updating object queues. Entries in the event table have associated timestamps, indicating when the kernel should react to that event. Objects have priority queues to serialize access to the set of methods exported by that object, where priority (order) depends on object semantics, scheduling criteria, and the timestamp and arrival orders of messages.

Typically, there would be a kernel for every processor in the network; in the current implementation, however, the prototype has only one physical kernel used for emulating a kernel per processor. We use a master real-time clock for the entire system (using abstract real-time units); the kernel is responsible for updating this clock, and all events are timestamped.

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. A store-forward mechanism, similar to SUPRA-RPC [47], has been adapted in the implementation to maintain the consistency of the values of out parameters at the conclusion of the call, when the execution of the callee is resumed, and restoring old state after preemptions.

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 timestamp 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.

7.2 Implementation of the Speculative Execution Transformations
To apply the compiler transformation rules for speculative execution, discussed in Section 5, requires the control graph,
a data dependence representation and the call graph. 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. 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. This makes the granularity of the condition big enough to enable the transformations. 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 topdown, 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.

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. Nonblocking 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 7.1, 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 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 body includes the code to be speculatively executed. The formation is performed by creating a new method whose line numbers facilitates carrying out the action part of the block. The new speculative methods so that the kernel can recognize them. The kernel will not update the clock until validating the speculative execution. 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 will consider the parallel execution of the speculative action and the remote execution of the call 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 handles only “no later than” timing constraints. Future extensions include verifying other types of constraints.

7.3 Experimental Results

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. Core algorithms 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 [5], [11], [12], [13], [44], respectively, have been written in CRL. The size of the resultant CRL programs ranges from 200 to 1,500 lines of code. Each application was compiled and analyzed for static timing behavior. The generated timed intermediate code was linked with the kernel and the runtime 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 [21] (without repeaters). While 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 section. 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).

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

---

5 In the current implementation no special treatment was performed to handle true dependence. Thus, the presence of true dependence causes the violation of the dependence preconditions and inhibits speculative execution.
applying the speculative transformation rules to the real-time programs mentioned above are shown in Table 2. The table reports the number of possible opportunities, feasible application of the 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 conditions in the applications as else is not counted.

Generally, the success rate can be seen to be highly influenced by programming style. Numerous 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, in the use of programmer annotations we see an interesting possibility to be tried in the future.

The effect of speculative execution on performance for the above programs is shown in Table 3. Speedup is measured as the percentage of the reduction in the program 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 section, 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 should scale with size of program. Better performance enhancements are expected for larger programs due to greater modularity.

8 CONCLUSIONS

While the use of speculative execution can, in general, degrade worst-case performance and complicate timing analyses for hard-real-time systems, we have shown that there are opportunities for safe use of these techniques. We have provided guidelines for identifying opportunities at compile-time and including these in generated code. To automate such analysis, we have presented a set of compiler transformation rules for speculative execution that preserve semantic correctness as well as timeliness.

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. The platform is based on a new object-oriented real-time language. The language and its runtime 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.

ACKNOWLEDGMENTS

The authors are indebted to generous support provided for this work under the United States Office of Naval Research under grants N00014-92-J-1367 and N00014-93-1-1047; and the United States Naval Surface Warfare Center under grants N60921-93-M-1912, N60921-93-M-3095, N60921-94-M-1250, and N60921-94-M-1426. A significant contribution to the implementation of the simulation has been made by Behrooz Ghahtyazi of Fairleigh Dickinson University. The authors are very grateful for discussions on real-time imaging with Purnendu Sinha, and for the many constructive comments made by members of the real-time computing laboratory and the anonymous reviewers.

At the time of submission of this paper, Dr. Alexander D. Stoyen was with the Real-Time Computing Laboratory, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, New Jersey.

REFERENCES


M. Younis, PhD dissertation, Dept. of Computer and Information Engineering, New Jersey Institute of Technology, 1996.


Mohamed F. Younis received the BS degree in computer science and the MS degree in engineering mathematics from Alexandria University, Egypt in 1987 and 1992, respectively. In 1996, he received his PhD degree in computer science from the New Jersey Institute of Technology. Dr. Younis is currently a research scientist at AlliedSignal Inc., Advanced Systems Technology Group, Columbia, Maryland, where he is leading multiple projects for building integrated fault tolerant avionics. He is also participating in the development of the redundancy management system, which is a key component for the vehicle and mission computer currently under development for NASA’s X-33 space launch vehicle. His technical interests include fault tolerant computing, system integration, distributed real-time and complex systems, and compiler-time analysis. Dr. Younis has served on many different committees and has numerous publications in his research field. He is a member of the IEEE and the IEEE Computer Society.

Grace Tsai received her BA degree from Tamkang University, Taiwan; an MS degree from the University of Missouri at Rolla; and her PhD degree from the University of Missouri at Rolla in 1994. Dr. Tsai’s dissertation topic was “Providing Assurance for Responsive Computing Systems.” She is currently an assistant professor of computer science at Fairleigh Dickinson University. She is also a visiting assistant professor in the Real-Time Computing Laboratory at the New Jersey Institute of Technology. Dr. Tsai’s research interests include formal methods in real-time systems, compiler transformation for speculative execution, parallel processing, and fault tolerance. She is a member of the IEEE and the IEEE Computer Society.

Alexander D. Stoyen received his PhD degree in computer science from the University of Toronto in 1987. Upon graduation, Dr. Stoyen joined the IBM Thomas J. Watson Research Center as a research staff member. In 1990, he became an assistant professor of computer science at the New Jersey Institute of Technology, where he built the Dependable Real-Time Systems Laboratory. When Dr. Stoyen left NJIT in January 1999, he was an associate professor of computer science and electrical and computer engineering. Currently, he is a senior director of computer science and director in the Center for Management of Information Technology at the University of Nebraska, Omaha. Dr. Stoyen has held visiting, consulting, and advisory appointments at universities, corporations, and other organizations throughout North America, Europe, and Asia. He has contributed significantly to a number of key technological concepts in real-time systems and has published more than 100 articles in journals, books, and conference proceedings. Dr. Stoyen has served as organizer of many professional events, chair of IEEE technical committees, and editor or guest editor of book series and archived peer-reviewed journals, including Computer, Real-Time Imaging, and Real-Time Systems. He was also founding CEO and president (until January 1999) of 21st Century Systems, Inc., a company researching prototyping and developing revolutionary tools for construction and operation of complex, large-scale software applications. Dr. Stoyen is a senior member of the IEEE and a IEEE Computer Society Golden Core member.