Toward Compiler Optimization of Distributed Real-Time Processes

Mohamed F. Younis*, Thomas J. Marlowe†, Grace Tsai‡ and Alexander D. Stoyenko*

* *AlliedSignal Inc., Microelectronics and Technology Center, Columbia, MD 21045, USA.*

‡ *Department of Computer Science, Fairleigh Dickinson University, Teaneck, NJ 07666, USA.*

† *Department of Mathematics and Computer Science, Seton Hall University, South Orange, NJ 07079, USA.*

* *Real-Time Computing Laboratory, CIS Department, New Jersey Institute of Technology, Newark, NJ 07102, USA.*

Abstract

Compiler optimization techniques have been applied to facilitate development and performance tuning of non-real-time systems. Unfortunately, regular compiler optimization can complicate the analysis and destroy timing properties of real-time systems. This paper discusses the difficulties of performing compiler optimization in distributed real-time systems. An algorithm is presented to apply machine-independent compiler optimization safely to distributed real-time systems. The algorithm uses resources' busy-idle profiles to investigate effects of optimizing one process on other processes. A restricted form of resource contention is assumed to simplify the analysis.

1 Introduction

Real-time systems developers do not have a uniform view of compiler optimization (and related techniques) and transformation. Many prefer to disable optimization, since, although these techniques have been applied successfully to non-real-time systems [9], they can destroy safety guarantees and deadline satisfaction in real-time systems. Others assume that preschedulability optimization of individual processes for improved average-case performance will not usually have negative effects on feasibility.

However, real-time applications have been growing substantially in size and complexity in recent years. As has been seen even in the non-real-time community, size and complexity make it ever more difficult to write hand-optimized code. On the other hand, the scale of the application, the increasing use of local timing constraints, and the need to use more powerful and less localized transformations make it ever more likely that standard optimization, particularly if designed to improve average-case performance without attention to worst-case execution time (WCET), may result in violations of timing constraints. Since proper transformation can also sometimes transform programs which may not meet constraints/deadlines, or which result in timeouts, into deadline-satisfying programs, safe optimization should be a priority in real-time systems.

In addition, safe compiler optimization can benefit even existing and feasible real-time programs. 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 system tasks or users, or for monitoring or debugging, but this may also make the programs more robust in the face of faults or unpredictable system overload, as suggested by the scheduling results of [1].

There has been an increase, during the past few years, in the use of distributed computation in the implementation of complex real-time applications, such as patient monitoring, avionics and flight control. Thus, transformations must consider other complicated issues such as synchronization and shared resources. Although performing safe compiler optimization will not extend the deadline of a process, it can affect the timing behavior of other processes. Consider, for example, the code in *Figure* 1 which consists of a loop followed by a call to a critical region `crit(R)`.

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 create a contention for a shared resource, causing an unpredictable delay and may cause, as a result, another process to miss its deadline. Assume that before optimization a process A will make a request to a resource R after other processes, say, process B and C have been serviced, as in *Figure* 2 (a). After optimization of process A, the call is reached earlier, so process A will compete with B

** This work was done under funding from the Office of Naval Research (grants N00014-92-J-1367 and N00014-93-1-1047) and the National Science Foundation (grant CCR-9402827) at the Real-Time Computing Lab at NJIT where the first author was a doctoral student, the second and third are visiting faculties and the fourth is a regular faculty. The authors are indebted to the many constructive comments made by the anonymous referees as well as by members of Real-Time Computing Lab. at NJIT. The authors also acknowledge the help they received from B. Ghahyazi in conducting the simulation.

```
   ORIGINAL                          OPTIMIZED

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

Figure 1: A real-time optimization

and C and may come ahead of process C in the resource queue, *Figure* 2 (b). Thus process C may wait longer in the queue, causing it to miss its deadline.

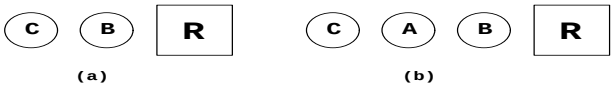

Figure 2: Optimization may disturb access order

In [5, 14], we show, in the absence of resource contention and inter-process dependence, that it is possible to safely apply optimization and/or parallelization to single real-time processes, improving WCET, or improving average-case performance without degrading WCET. In [14], we present a number of such rules, and prove their safety. Related results by other researchers are presented in [2, 7]. However, in the presence of resource contention, synchronization, or other inter-process dependence, it is hard to guarantee that an optimization in one process will not negatively affect the other processes in the system. Access time of resources can be easily disturbed, typically because of a reordering of resource requests, and other processes may anticipate extra delays. This motivates our study of performing safe optimization transformations for multiprocess real-time systems.

In the next section, we study the complexity of the problem as well as some simplification approaches. *Section* 3 provides a summary of previous work. *Section* 4 discusses our assumptions concerning contention. Performing multiprocess analysis is illustrated in *Section* 5. The optimization algorithm is discussed in *Section* 6, followed by a presentation of simulated results. Finally, we conclude with a summary and future research.

2 Reducing Problem Complexity

Safety of compiler transformations in a communicating multiprocess real-time systems requires two sets of tests. First, effects internal to the process are to be studied; the process deadline should not be extended along the worst execution path. Second, resultant changes in the execution behavior of other processes must be analyzed. This analysis checks adherence of processes to their critical timing constraints under all possible execution orders compatible with the scheduling discipline in use. Such analysis, commonly referred to as *schedulability analysis* [10], must be applied even in the absence of optimization.

However, finding precise solutions considering contention and branching is, in general, an NP-complete problem, and it can add significantly to the cost of program compilation. The NP-completeness arises, in particular, from the combinatorial explosion of possible execution orders in case of communicating processes, especially when requests occur in conditionally executed code. As a result, schedulability analysis can either be (1) exact and efficient analysis of single process or multiple processes of simple form, or with highly constrained interactions [7, 8], (2) highly imprecise though efficient analysis of multiple process programs [4], or (3) nearly exact though highly inefficient analysis of some multiple processes [10]. To combat some sources of combinatorial explosion, there has been work to reduce the cost of precise schedulability analysis, such as [11, 12].

Detecting shared resource access time is a key difficulty in multiprocess analyses of real-time systems. The response time depends on the time of the call and the resource queue size during the call. Every combination of possible queue orders for requests from various processes needs to be considered, which makes the analysis exponential in the number of processes. Simplification techniques have been developed to perform that kind of analysis, either by restricting the resource access model [11], or by assuming a smart scheduler [3], as illustrated in *Section* 4.

Our approach uses the restricted resource contention model of [11] to predict the response time of resource requests. We rely on extracting busy-idle profiles of shared resources during compilation. Consulting the resource busy-idle profile before applying the code transformations, we can predict effects of optimizing a process on other communicating processes.

3 Previous Work

In addition to [3, 5, 11], related work includes work in real-time optimization, and work on simplifying schedulability analysis. We consider two categories of transformations: (1) to improve overall performance or enhance schedulability of processes, (2) to reduce the complexity of schedulability analysis.

Code motion is used in [2] to re-organize the code of a sequential process to meet timing requirements. Forking new processes to speculatively execute blocks of the code without destroying timeliness in case of rollback is discussed in [14]. Basically, they only consider single process transformations; none consider the effect of transformations in a multitask system.

Busy-idle profiles of resources are used in [3] to expose the potential for parallelism across tasks to the scheduler. Here, we use busy-idle profiles to check the effects of transformations on other processes. We try to speed up individual processes without affecting others, rather than simply informing the scheduler.

A polynomial-time code transformation to simplify schedulability analysis of real-time programs is pre-

sented in [11], where a restricted form of shared resource contention of processes is assumed to simplify the analysis. Effects of conditional branches on complexity can be reduced if by inserting fixed delays in one branch, it can be transformed to a timewise replica of the other branch. Conditional linking has been shown to enable additional transformation in [12].

4 Calculating Resource Access Time

As discussed in *Section* 2, a major difficulty in applying compiler optimization in multiprocess real-time systems is the prediction of effects on shared resource response time. Precise analysis is time-consuming because all combinations of shared resources requests need to be considered for an accurate prediction of worst-case queue sizes. For example, with N parallel processes making mutually exclusive requests to a shared resource, the resource queue size can range from 0 to $N-1$. However, not all processes will make those requests at the same time. An accurate queue size can be calculated considering all (possibly exponential number) execution paths of the processes. Analysis, similar to [4], and transformations, like [11, 12], can be applied to decrease the number of paths to be considered.

Our approach to calculate shared resource access time requires the following two steps:

1. Extract for each process (as precisely as possible) its access profile with respect to every shared resource in the system.
2. For every shared resource, combine access profiles of all processes to build the resource's busy-idle profile.

For simplicity, assume that resources can be partitioned, so each call uses exactly one of the partition sets. Resources' processors are independent. Moreover, resource requests are not nested and neither can be removed by optimization nor contain optimizable code. Thus, the service time of a resource will not change due to optimization. We assume a set of periodic processes which are assigned to different processors. Consequently, it is sufficient to study the behavior of the processes in the *Least Common Multiple* of process periods. In the following subsections, we discuss how to perform these two steps.

It is relatively simple to extend this to the case in which resources are not partitioned, provided that resources are still on independent processors and there are still no nested calls, and that a call which uses resources in multiple sets must acquire them all to be served, and holds them all for the duration of the call. However, each resource can be released after the call at the end of its own busy interval.

Also, while in principle different frame lengths for processes can result in a need to optimize many distinct versions of a process, we expect this to seldom be a serious problem in practice. Most often, only a few such lengths occur in a given example, except perhaps for a sequence of lengths in which each larger length is a multiple of the previous, so there will typically be only a few versions of a process to consider. Even when there many instances, some may be equivalent relative to resource requests (by all processes) in their lifetime, and others may occur in low-contention settings, where optimization is both easy and less important.

4.1 Access Profiles for Resources

To extract a processes' access profile for every resource, the compiler must perform timing analysis of every process. This timing analysis provides a safe static estimate of the execution time of programs. In real-time systems, the important metric is the worst-case execution time. Timing information is extracted during semantic analysis. Various techniques, like loop unwinding and call in-lining, may be used to obtain estimates of the execution time of loops, recursive calls, etc. Architecture-dependent analysis can be used to tighten those estimates [6]. The output of this step is a timing profile of requests being initiated from each individual process, as in *Figure* 3. There are many algorithms proposed in the literature which can be used to build such profiles, for example [3].

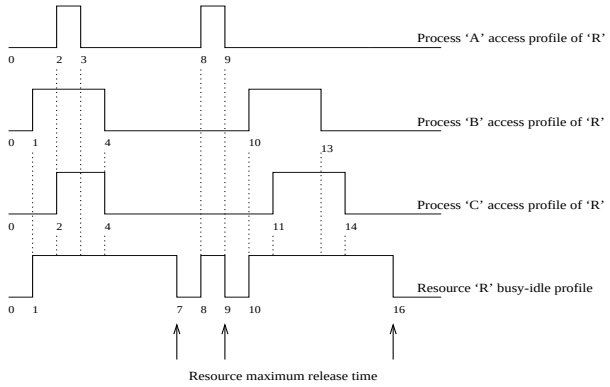

Figure 3: Access profiles of a shared resource

4.2 The Resource Contention Model

Once we extract resource access profiles per process, the next step is to build the resource's busy-idle profile. Because, at compile-time, we have no clue about the order of requests from various processes, one of two assumptions can be used. We may assume a certain service order, relying on the scheduler to respect that assumption. Alternatively, we can be conservative, using a form of restricted resource contention. We adopt the second approach, restricting shared resources' response time, using the following assumptions:

1. Whenever there is a queue, all processes involved (including the first, which claimed the resource without waiting) are released at the same time.
2. There is a statically-known fixed release time (and thus, a corresponding queue size) for every resource busy interval.

For example in *Figure* 3, combining the resource R access profile for processes A, B, and C, will result in the illustrated busy-idle profile for R. Notice the corresponding release times for the various requests. Even though process B makes the request at time 1, the release time of all requests from A, B, and C will be at 7, reflecting the sum of the service time of the three requests. An algorithm for calculating the resource access profile can be found in [15]. The algorithm considers *Least Common Multiple* (LCM) of periods for a set of periodic processes. If a process contains unresolved resource requests on multiple branches, the algorithm assumes that all of them may occur.

5 Multiprocess Safety Analysis

Opportunities for optimization within a single process depend on whether we have access requests for shared resources. In the absence of critical sections, it is always safe to apply compiler optimization techniques as long as we are not extending the execution time along worst-case path. We call this property *local safety*. However, in the presence of critical sections, safety of the transformation depends on not worsening shared resources' busy-idle profiles. We refer to that property by *multiprocess safety*. Thus, a transformation is safe if it is locally safe, and multiprocess safe. Consider the example in *Figure* 1, in the absence of the call $crit(R)$, moving the loop invariant out of the loop is always safe. However, it is not guaranteed that it is safe in the presence of the call unless we prove that it is multiprocess safe. Removal of unreachable code is necessarily multiprocess safe, but essentially no other transformation is always safe.

Our approach, as we illustrate, is to inject delay statements in the code rather than actually optimize during local analysis, to avoid changing the access profiles of the process with respect to each resources. Then, we perform multiprocess analysis to defer and eventually remove these delay statements without destroying the multiprocess safety property.

The analysis of effects of a single process optimization on other processes can be performed using shared resources' busy-idle profiles. In this section, we discuss the multiprocess safety property of compiler transformations. We say a request lands in an idle interval if its service does not overlap any busy interval; it lands in a busy interval if it overlaps other services. Transformations may affect the resource busy-idle profile by moving a request:

1. out of an idle interval into an idle interval,
2. out of a busy interval into an idle interval,
3. from one busy interval into the same interval,
4. from one busy interval into another busy interval,
5. from an idle interval into a busy interval,
6. causing a merge of two busy intervals.

In cases 1 and 5, "idle interval" should be taken to mean that the current request is the only one in the current busy interval.

The first is generally safe while cases two and three can be applied safely with care. Moving a request from an idle interval to the same or different idle interval is always multiprocess safe. Such optimization will not introduce contention. On the other hand, moving the request within the same busy interval will not affect the release time, according to our model, of all the processes sharing that busy interval unless the entire interval will be shifted forward. Shifting the busy interval forward will decrease the release time of all requests in this busy interval, which may affect other processes. In this situation, the amount of shift can be compensated by inserting exactly a compensating amount of delay past the resource request in each of those processes. The second case is ideal. Moving a request from a busy interval into an idle one will not only accelerate the execution of the optimized process, but also will reduce shared resource contention and speed up other processes competing for that resource, and may actually split the busy interval. Consider the example in *Figure* 4 (a), where the resource R busy-idle profile has been produced by combining processes A, B, C, and D access profiles. Assume we successfully have optimized the code of process A, so that its request at time 8 can be initiated at time 7. This will split the busy interval (8,15) into two intervals (7,9) and (10,15), as shown in *Figure* 4 (b). Now the response time for the request from processes A and B will be faster, due to reduced contention. However, the decrement in the release time for the other processes should (at least temporarily) be compensated by inserting delay after the request. In our example, a delay of 6 units should be inserted after the call made by B, and one of 7 units after the call in A.

The fourth and fifth cases are not in general multiprocess safe. Moving a request from a busy interval to another busy interval needs more careful analysis. This case can increase the contention in the target busy interval and extend the release time. On the other hand, it may be beneficial to reduce contention in an interval as long as the increase in the release time of the target busy interval is acceptable. In *Figure* 4 (a), if the request after optimization of process A becomes at time 6, this will extend the release time of the previous requests to 7 instead of 6, while enhancing the response time of the request from B at time 8. Such cases require checking all the processes that will suffer and gain from that optimization, which may lead to exponential analysis. However, there are obvious unsafe situations, such as linking two busy intervals (case 6). For example in *Figure* 4 (a), moving the request of process C from time 10 to time 6 will link the two busy interval. There are nonetheless two safe special cases. First, if all the requests in the target busy interval were originally in the same source interval. Second, when all processes participating in the target busy interval have a delay past the request large enough to accommodate the extension in the release time. This subsumes the first case, since shifting previous requests out of the current interval creates such delays. On the other hand, moving a request from an idle interval into a busy interval will introduce new contention while only one process will gain. Thus, case 5 can be avoided or handled similar to case 4. Sometimes, it may be highly desirable to

decrease the execution time of a process to meet its timing constraints, as long as the other processes can tolerate delays due to increased contention.

Applying multiprocess safe transformations may split busy intervals. Moving one request from one of these subintervals to another should not be considered unsafe; even merging them again should be considered safe. Therefore, the resource busy intervals should be identified. Splitting an interval by a safe multiprocess creates two subintervals with the interval id. This allows application of safe subcases of 4, 5, and 6.

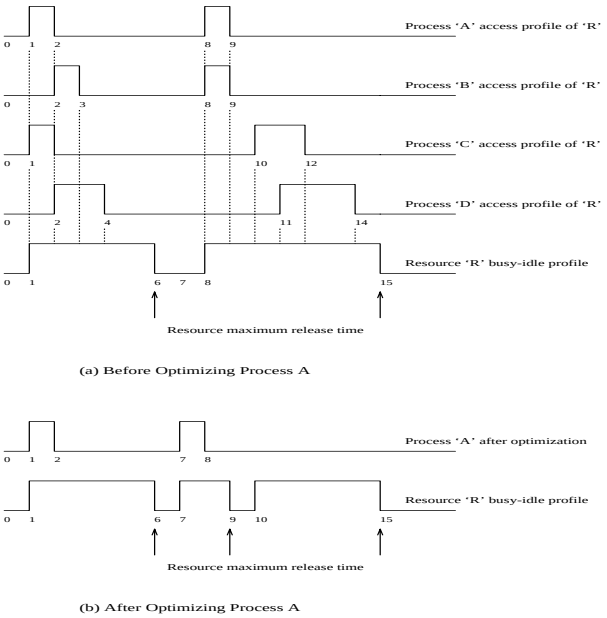

Figure 4: Effects of optimization on response time

In the next section, we provide an algorithm to perform optimization without destroying either local or multiprocess safety.

6 The Optimization Algorithm

Our algorithm for applying machine-independent compiler optimization safely for distributed real-time systems is composed of two phases. In the first phase, we consider each individual process in isolation. Code improvement transformations will be applied safely, in the sense of [5], for individual processes. However, in all cases gains in performance are compensated by injecting delays to preserve the shared resource access profile. These delay statements are the seeds for delay propagation in the second phase.

For simplicity, we assume all timing constraints are absolute max constraints, relative to the start of the process frame. Timing constraints can be applied to any statement. In our algorithm, we assume by default that precisely the set of shared resource requests and the end of the process are observable. However, the algorithm can be extended easily to accommodate other timing constraints. Again, according to our restricted resource contention model, whenever there is a queue, all processes involved (including the first, which claimed the resource without waiting) are released at the same time. Multiprocess analysis is performed in the second phase of the algorithm. Delay statements are pushed toward the end of processes while consulting the busy-idle profiles of shared resources. A cleanup step is performed at the end of this phase to remove delays reaching the end of processes. However, we may need to perform different delay shift and removal optimizations in each instance of every process in LCM. Successfully removed delays in the first instance does not necessarily mean it is safe to remove them in another, due to differences in contention for resources. We may thus need to clone different instances of a process within LCM on the same processor. Alternatively, a set of directives can be generated to the scheduler to report the difference between various instances. The algorithm outline is shown in *Figure* 5. In the following subsections, we explain the two phases of the algorithm in more details.

```
// Algorithm to perform compiler optimization
// Extract process's access profiles for shared resources
For i = 1 To Number_Processes
    Extract shared resources access profiles
EndFor
// Build shared resources busy-idle profiles
For i = 1 To Number_Resources
    Combine access profiles of Resource_i for all processes
EndFor
// Single Process Optimization
For i = 1 To Number_Processes
    Optimize Process_i with injecting delays
EndFor
// Multiprocess safety analysis
apply algorithm Shift_delay
// Single Process cloning if necessary
For i = 1 To Number_Processes
    clone the Process_i if necessary on the same processor
    or generate directives for the scheduler
EndFor
```

Figure 5: Safe compiler optimization algorithm

6.1 Single Process Optimization

Most optimization techniques can be viewed as removing unnecessary code, moving code, or replacing it by faster equivalent computations. The common goal is to reduce the execution time of programs, and optimization usually results in performance speedup. For example, code invariant motion out of a loop reduces the loop body for faster execution. Common subexpression elimination removes useless computation. So, we argue that optimization can be viewed as a transformation to wasteful code (or delay), followed by elimination of that redundancy. Our approach is based on deferring elimination of delays until the multiprocess safety property is guaranteed.

In the algorithm (*Figure* 5), we require that any time gained by the optimization be compensated by injecting delays, so as not to affect process access profiles with respect to shared resources. After the first

phase of the algorithm, the timing behavior and resource access of every optimized process remains the same. The time, relative to the beginning of the process, of all access requests for shared resources will not change; consequently, busy-idle profiles of resources will not be affected by these transformations. In the second phase, we try to move and eventually eliminate injected delays in individual processes without affecting the timeliness of the other processes.

This phase generates for every process an array of points of interests and their times relative to the start. Points of interest are locations in the code that contains either a delay, a resource request, a split of a conditional, a merge of a conditional, or the end of a process. These points will be used by the second phase to correctly propagate delays towards the end of the process. We refer to this array as $Test\_Points$. Delay points includes delays which have been introduced by other transformations, for example [11]. Resource request points contain the resource busy interval id. We extend the $Test\_Points$ list by replicating the points of the first period, a number of times equal to the number of instances of the process in LCM (LCM/period size). The idea is to work on this array and clone the process on the same processor, if necessary, at the end.

6.2 Multiprocess Analysis

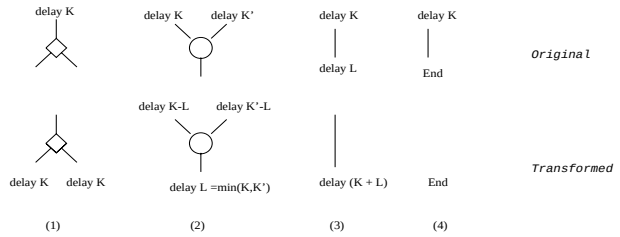

Figure 6: Rules for delay propagation (easy cases)

In the multiprocess analysis phase, we try to defer and then remove delays inserted in the code during the first phase. The idea in this phase is to shift the delay statement(s) in the process code towards the end, using a greedy approach. Delays reaching the end of the process may safely be removed. Shifting a delay will be performed in steps. Every step we try to move a delay beyond a call to a shared resource, so that the call will be made earlier. We consult the resource busy-idle profile(s), for the call. If it is multiprocess safe to make the request earlier, we shift the delay. We continue until the end of the process code, removing delays reaching there. This operation is assisted by the array $Test\_Points$ generated by the first phase.

Delays can be propagated toward the end using the rules in *Figure* 6: (1) In a split point (conditional), delays can be propagated in both branches. (2) In a merge, we propagate the minimum of the delays in the branches. (3) Consecutive delays can be combined. (4) Delays reaching the end can be safely removed. Propagating a delay beyond a resource request needs further multiprocess analysis, checking the busy-idle profile of that resource. If a request appears in a condition, delays that can be shifted successfully beyond this condition need to be propagated to both branches.

The outline of algorithm of the multiprocess safety check can be found in [15]. The idea is to set up a simulated clock that can progress up to the LCM. The clock advances to the earliest delay point in all processes, and so on. This delay may be propagated if it is safe. Note that we work only on the $Test\_Points$ generated from the first phase. A delay is shifted forward by successfully applying the appropriate rule in *Figure* 6. In case of a resource request, the algorithm tries to remove as much delay as possible from before the request to a compensating delay after the process. The algorithm checks the safety of moving one delay unit at a time by testing the impact on the busy-idle profile of the resource. The test will need to recalculate the busy-idle profile of the resource; the algorithm relies on checking the location of the new request in the current busy-idle profile of the resource. The new request time and the service time are used to relate the new location to the current profile and predict the effect on the response time of that request as well as other processes. The classification of the effect on the resource busy-idle profile, discussed in *Section* 5, is performed according to the table in *Figure* 7 (some subcases must be handled with care: case 3 if the start of the interval shifts, case 4 and 5 when there are covering delays). Every resource busy interval will have a flag indicating whether all requests in this interval were originally located in a single resource busy interval. This flag will be set to *mixed* when a request from another interval (interval with different id) is to be added. Using such flag enables handling safe subcases of 4, 5, and 6 when the target interval is composed of requests previously moved from the current interval. *Ideal release time* is the response time in absence of contention. Note that in cases 4 and 5, we are not performing an exhaustive analysis for all processes anticipating extra release time due to contention, which in fact may need exponential time; we are only detecting simple subcases which are proven to be safe by simple analysis. For a more detailed discussion of the multiprocess safety analysis algorithm and analysis of its complexity, the reader can refer to [15].

New Call Time	Ideal Release Time	Status	Case
Same interval	do not care	Safe	1,3
Idle interval	Same interval	Safe	2,3
Different interval	Same interval	Unsafe	6
Different interval	Idle interval	Investigate	4,5

Figure 7: Multiprocess safety classification

7 Experiments

The success of applying our algorithm is application dependent. We may be able to remove all, some, or in the worst case none of the delays. In the this section, we examine the applicability of our algorithm

through a preliminary experiment based on simulation. The simulation design is largely based on samples of real-time programs [12, 14]. Real-time processes are assumed to run on multiple processors which are connected by a backplane bus (e.g. avionics). The experiment captures the success of our algorithm to adapt performance gains achieved by single process optimization. In addition, the experiment tries to study the impact of the frequency of calls to shared resources on the applicability of our approach. In this section, the design of the simulation is explained and the experiment results are discussed.

7.1 Design of Simulation

The experiment consists of the following steps:

(1) Generating Workloads: A program is a group of statements selected out of the following; IF, WHILE, ASSIGNMENT, CALL, BLOCKING_CALL, READ, and WRITE with frequencies 10%, 10%, 35%, 20%, 5%, 20% respectively. These frequencies are assigned based on experience with real-time programs [12, 14].

Both READ and WRITE use buffers and are considered non-blocking. Calls can be blocking (BLOCKING_CALL) to access a shared resource, or non-blocking (CALL). Loops and if statements are not primitive statements, in the sense of containing more than one statement. 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.

(2) Assigning Times to Statements: For a primitive statement, we assume that the execution time is proportional to the number of variables involved in that statement. Consequently, the execution time of a primitive statement can be computed by multiplying the number of variables involved by a constant whose value is based on a translation to the Intel 80386 instruction set.

(3) Calculation of WCET and deadline: The generated program control flow graph is analyzed to calculate an upper bound on the execution time. For primitive statements, WCET is the assigned execution time at the previous step. For conditional statements, the time of the longest path is used as WCET. The upper bound of the loop index is used to calculate the WCET of the loop.

In this simulation, we assume that the deadline is exactly WCET. Therefore, we use WCET of processes to calculate the least common multiple (LCM). Next, we inject delay statements and adjust the WCET.

(4) Injection of DELAY Statements: We conservatively assume that single-process compiler optimization can speed programs up by a percentage up to 10%. We randomly select a speedup percentage S in the range [0,10]%. DELAY statements are inserted with a total delay size S * WCET. These DELAY statements are inserted randomly throughout the program. The worst-case execution time, calculated in the previous step, is adjusted by multiplication to the a factor of $(100 + S)$%.

After delay insertion, the program control flow graph is traversed and the array *Test_Points* of points of interest is generated, as discussed in *Subsection* 6.1.

(5) Constructing Resource Access Profiles: As discussed in *Section* 4, to build the resource access profile for each process, first we apply the analysis of [11] to balance access to shared resources along branches of conditionals. Loops need to be unrolled (which might be done in the previous step while calculating WCET) to calculate the call time to the shared resource.

(6) Building The Busy-Idle Profile: Combining resources access profiles of individual processes, we construct busy-idle profile of shared resources. The restricted resource contention model, described in *Subsection* 4.2, is used to compute the boundaries of busy intervals. Note that a busy interval may contain several calls from different processes. The start time of an interval is the start time of the earliest request and release time is the sum of the start time and service times of all requests in the that interval.

(7) Multiprocess Analysis: Finally, multiprocess analysis tries to push delays to the end of programs. Simulation results are described in the next subsection.

7.2 Evaluation

We run experiments with different number of processes and resources. The table in *Figure* 8 shows the results sample runs including the number of processes, the number of shared resources, the percentage of blocking calls in programs, the percentage of injected delays relative to WCET, and the safe and unsafe cases found during the multiprocess analysis.

As expected, the increase in the number of processes affects the number of successful cases. The busy intervals in resources profiles become close to each other. Unsafe subcases of 4 and 5 appear more frequent. The table also indicates that the increase in the number of shared resources accesses has the same effect. Case 2 happens with the highest rate, which indicates the effectiveness of our algorithm on reducing contention in distributed real-time systems. According to the algorithm, resource waiting time will be added as a post-access delay which significantly reduces WCET (*Figures* 9 and 10).

Figures 9 and 10 also show that the reduction in WCET scales with the size of the injected delays (single process compiler optimization). WCET decreases with the increase in the number of requests. However, the amount of the decrease diminishes and eventually will saturate because the increase in resources requests makes boundaries of the resource busy intervals closer and consequently harder to shift delays. On the other hand, small number of requests shrinks the effect on WCET, since resources contention is less significant.

8 Conclusion and Future Work

While single process optimization can be performed safely, it may affect other processes by introducing contention for shared resources. We have developed an algorithm to apply code improvement optimization safely in distributed real-time systems. The algo-

process	resource	request	delay	case 1	case 2	case 3	case 4+5	safe	case 6	case 4+5	unsafe
10	5	3%	5%	57	79	21	10	167	1	67	68
10	5	5%	5%	92	166	29	48	335	3	67	70
10	5	8%	5%	174	288	47	72	581	5	480	485
20	10	10%	10%	194	150	43	75	462	80	452	532

Figure 8: Summary of safe and unsafe cases

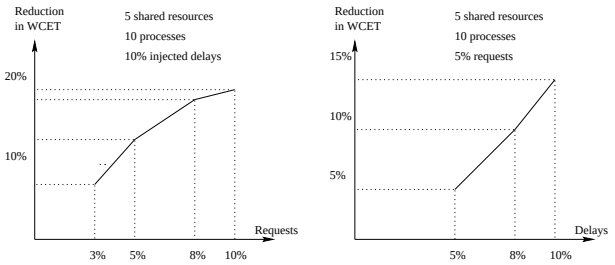

Figure 9: WCET versus requests and delays

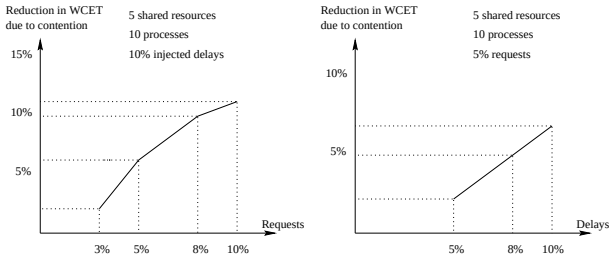

Figure 10: Reduction in resource contention

rithm relies on a model of resource use and on scheduling with a restricted resource contention model. The approach is based on applying machine-independent compiler optimization in two phases. In the first phase, we perform the transformations, compensating the enhancement in performance with delays. In the second phase, we try to remove delays when this can be proved to be multiprocess safe. Simulated results indicate that we can not only enhance the worst-case execution time of distributed real-time processes, but also reduce resource contention.

Currently, we are extending our resource model to allow for resource optimization. We intend to integrate our algorithm in a platform for complex real-time systems. We plan to test the applicability of compiler optimization techniques in real applications.

References

[1] S. Baruah, L. 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.

[2] R. Gerber, 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.

[3] R. Gupta, 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.

[4] D. Leinbaugh, 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.

[5] T. Marlowe, 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.

[6] K. Nilsen, Bernt Rygg, "Worst-Case Execution Time Analysis on Modern Processors," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, La Jolla, California, June 1995.

[7] V. Nirkhe, 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.

[8] 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.

[9] G. L. Steele Jr., et. al., "Fortran at Ten Gigaflops: the Connection Machine Convolution Compiler," *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'91)*, pp. 145–156, 1991.

[10] A. Stoyenko, V. Hamacher, R. Holt, "Analyzing Hard-Real-Time Programs for Guaranteed Schedulability," *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, pp. 737–750, August 1991.

[11] A. Stoyenko, T. Marlowe, "Polynomial-Time Transformations and Schedulability Analysis of Parallel Real-T ime Programs with Restricted Resource Contention," *Journal of Real-Time Systems*, Vol 4, No. 4, pp. 307–329, 1992.

[12] A. Stoyenko, T. Marlowe, W. Halang, M. Younis, "Enabling Efficient Schedulability Analysis through Conditional Linking and Program Transformations," *Control Engineering Practice*, Vol 1, No. 1, pp. 85–105, 1993.

[13] M. Wolfe, *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[14] M. F. Younis, "Safe Code Transformations for Speculative Execution in Real-Time Systems," Ph.D. Thesis, Department of Computer and Information Science, New Jersey Institute of Technology, June 1996.

[15] M. F. Younis, T. J. Marlowe, G. Tsai, A. D. Stoyenko, "Applying Compiler Optimization in Distributed Real-Time Systems," Technical Report CIS-95-15, Dept. of Computer and Info. Science, New Jersey Inst. of Tech., 1995.