(2)

Final Report

Utilizing Imprecise Results in Real-Time Systems

Kwei-Jay Lin
Jane W. S. Liu

October 1990

Utilizing Imprecise Results in Real-Time Systems

Contract No.: N00014-87-K-0827

Final Report

October 1990

Co-Principle Investigators:

Kwei-Jay Lin

Jane W. S. Liu

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Prepared for

The Department of the Navy
Office of Naval Research
Arlington, Virginia 22217-5000

PI Names: Kwei-Jay Lin and Jane W.S. Liu
Institution: Univ. of Illinois, Dept. of Computer Science
Phone Number: (217) 333-1424
E-mail Address: {klin, janeliu}@cs.uiuc.edu
Contract Title: Utilizing Imprecise Results in Real-Time Systems
Contract Number: N00014-87-K-0827
Reporting Period: 1 Sep 87 - 31 Aug 90

Productivity Measures

Refereed papers submitted but not yet published: 7

Refereed papers published: 12

Unrefereed reports and articles: 2

Books (and sections thereof) submitted but not yet published: 0

Books (and sections thereof) published: 0

Patents filed but not yet granted: 0

Patents granted: 0

Invited presentations: 2

Contributed presentations: 0

Honors received: 0

Awards/Prizes: 0

Ph.D. degree granted: 1

Graduate students supported: 2

Post-docs supported: 0

Minorities supported: 0

PI Names: Kwei-Jay Lin and Jane W.S. Liu
Institution: Univ. of Illinois, Dept. of Computer Science
Phone Number: (217) 333-1424
E-mail Address: {klin, janeliu}@cs.uiuc.edu
Contract Title: Utilizing Imprecise Results in Real-Time Systems
Contract Number: N00014-87-K-0827
Reporting Period: 1 Sep 87 - 31 Aug 90

Summary of Research Results

In a hard real-time system, all real-time tasks must complete before their individual deadlines. Since the execution times of real-time tasks may vary and the load level on a system may fluctuate, it is difficult to guarantee that all deadlines are met at all times. In our model, a result produced by a computation is said to be *precise* if the computation runs to normal termination. If an intermediate result is made available by the computation at the deadline even when it has not totally completed, the result is said to be *imprecise*. This project is concerned with the design and evaluation of programming primitives, scheduling algorithms and system supports for the imprecise computation model. The model allows early results produced by partially completed computations to be effectively used to ensure the timeliness of real-time systems and to achieve graceful degradation.

In the past year, we have looked into four issues related to real-time imprecise computations: timing constraint implementation, performance analysis, performance polymorphism, and scheduling parallelizable imprecise computations. In this report, we present the summary of research in each of these topics.

I. Implementing Timing Constraints

The programming language used to implement real-time applications should allow different types of timing requirements to be specified as well as providing ways to guarantee that the requirements will always be met. We have designed and implemented two real-time programming languages: FLEX and Real-Time Mentat. FLEX is targeted toward single-processor real-time systems, while RT Mentat is designed to run on multi-processor systems. In both languages, temporal constraints are used to express timing requirements. The scope of a constraint is defined to be a *constraint-block* (CB) which can be a statement, a group of statements, or the entire computation. As an option, exception handling may be specified for situations when the constraint cannot be satisfied. Each CB has four timing attributes: *start*, *finish*, *duration* (during which the CB is allowed to execute), *interval* (between successive executions of the block). These timing attributes are complete in the sense that they can be used to express all temporal requirements for the execution of a program.

The timing constraints in FLEX come in two varieties: the 'earliest time' and 'latest time' constraints. If these are applied with dependencies among different blocks, there are two fundamental ways of implementing relations among the events. Let us say that block A may not start until block B has finished. We may view this as a requirement to delay the execution of block A. Alternatively, we may view the constraint as a deadline on block B, and require that B be completed before A starts, causing a timing fault if this constraint is violated. Clearly these two constraints are synonymous in terms of temporal logic, but there is a world of difference in their behavior at run time. As a matter of programming style, we prefer to impose as few arbitrary deadlines as possible upon tasks. The former usage of causing a delay is to be preferred over the latter usage of imposing an additional deadline wherever possible.

To implement the complete structure of a constraint block, we need system primitives that maintain program context, track constraints on variables, limit execution time, and delay program execution. The

flow of control through a constraint block is as follows:

```
Constraint_Block E;
{
 Context _C_n;          // Establish context
 if ( _C_n.save() )
 {
  -- user's exception handler goes here --
 }
 else
 {
  -- declarations for non-temporal constraints --
  -- declarations for latest finish time --
  -- declarations for latest start time --
  -- code to delay until earliest start time --
  -- code to clean up latest start time structures --
  E.start = NOW();    // Record the start time

  -- body of the constraint block --

  -- code to delay until earliest finish time --
  -- code to clean up latest finish time structures. --
  -- code to clean up non-temporal constraints --
 }
E.finish = NOW();      // Record the finish time
}
```

Certain limitations are placed on the form of constraints in the current implementation. These restrictions derive from the fact that we have considered it desirable to avoid recursion in the constraint mechanism at run time, to ensure that the overhead will be predictable.

The first of these is that functions appearing in constraint expressions are assumed to be referentially transparent Ordinary mathematical functions, like *sqrt*() or *cos*(), will present no trouble. Functions are assumed, though, to return values that depend only on their arguments. If a function has side effects, or if it returns values that depend on some state of the program external to the arguments, that dependency will not be recorded in the constraint system and the results will be unpredictable.

The second restriction is that assignments to reference objects (pointers and array indices) appearing in constraint expressions are not checked. If a constraint expression contains a form such as $p \rightarrow value$ or $a[n]$, p and n will be assumed to be constant for the lifetime of the block. This limitation could be removed with some additional programming effort, if one is willing to accept the unpredictable overhead of a recursive constraint mechanism. An additional type of constraint, the reference constraint, would have to be implemented. A reference constraint would link a reference expression (p) with the variable that it designates ($p \rightarrow value$). When an assignment is made to p, the constraint would have to unlink the previous target of the reference from any constraints in which the reference appears, link in the new target, and then force the constraint to be reverified. This reverification could result in further steps, in the case of expressions like '$p \rightarrow q \rightarrow r$' or in certain examples of aliasing.

When a constraint depends on the value of a time interval, it is deemed satisfied only if it is satisfied for every time within the interval. For example, if a constraint takes the form, $start \geq A.start$, and the start time of block A is known only to be in the interval, $[t_1,t_2]$, then the constraint is satisfied only if $NOW \geq t_2$. This strict interpretation of constraints leads to some behaviors which may be unexpected. For instance, consider the constraint, $start \geq A.start - 10$, which says, 'this block may not start earlier than 10 units before the start time of block A.' In the absence of other information about block A's start time, this constraint will delay the start until A's start time is known, that is, until A has started. The -10 in the expression is therefore meaningless. (It is unclear what the semantics should be in this case.)

More unexpected, perhaps, is the behavior of *start*≤*A.start*−10. This constraint states that 'this block must have started by the time that is 10 units before the start time of block *A*.' In the absence of other information about block A's start time, this will cause a timing fault, as there is no guarantee that block *A* will start in the next 10 time units. Delaying execution until the start of *A* wouldn't help; in that case, since block *A* has already started while the current block has not, the constraint fails and the timing fault is detected. Once again, it is unclear what better choice of semantics could be made for this admittedly pathological case.

II. Performance Analyzer

One of the first issues for verifying the correctness of a real-time program is to determine its expected execution time. A number of projects have presented methods for determining the maximum time of a task. Their schemes operate by limiting the programmer to a restrictive set of control structures (e.g. bounded loops, no recursion), and calculating the number of times that each machine instruction in a task will be executed. They use some sort of underlying model of the target machine's hardware performance to calculate the expected execution time, given this information.

We have implemented an alternative to the above technique. Rather than attempting to determine the timing behavior analytically, the alternative technique measures the actual timing behavior and determines, from the measurement data, the parameters of a programmer-supplied model of the timing. The model can be very simple, since it represents the programmer's understanding of the program's timing behavior in the 'big-oh' sense. For instance, if a programmer expects the time consumed to be quadratic in the size n of a data set, he can specify that the time $t=An^2+Bn+C$ for some parameters A, B, and C. It is then up to the measurement system to determine the values of A, B, and C. In this way, the precise timing characteristics of the program can be derived.

To support timing measurement and modeling, the FLEX language is augmented with a new directive, *#pragma measure*. This directive both directs the compiler to generate code to measure the time or resources consumed by a block of code and provides the parametric model for analyzing the measurement. In addition to specifying the expected complexity function, the pragma allows an optional *safety* clause to be specified a safety factor - a number by which to multiply the computed time when determining the time to allow for a computation. The safety factor allows the programmer to specify that the system should, for example, always provide at least ten percent more time than the worst case seen so far. The following example shows the pragma syntax.

```
void isort (register int* x, register int n)
#pragma measure mean duration defining A, B, C in (A*n+B)*n+C  safety 1.1
{
 for (register int i = 1; i <= n; ++i) {
  register int y = x [i];
  for (register int j = i-1; j >= 0 && y < x [j]; --j)
   x [j + 1] = x [j];
  x [j + 1] = y;
 }
}
```

The *#pragma measure* directive causes the generated object code to include software probe points for measuring the block of code. For each execution of the block, the program records the location of the *#pragma measure*, the contents of all the free variables, and the value of the performance statistic being measured.

Once the program has been run, possibly a number of times, to collect the measurement data, a measurement analysis program is run. This program determines the best fit of the parameters to the observed run time. It produces a report that describes the values thus determined, and gives a confidence level (determined using the χ^2 statistic) for the model. The process of determining the model parameters given the observed time and resource data is a curve-fitting problem; we describe it statistically as a χ^2 minimization problem. For each observation i in a *#pragma measure* statement, the system records the

observed quantity (the amount of time or other resource consumed) y_i. It also records the contents of the free variables $\mathbf{v}_i=[v_{i1},\cdots,v_{im}]$ in the expression. As an initial data reduction, groups of observations having the same values for the free variables are replaced by single records giving the free variable contents $\mathbf{v}_i$, the mean value of the observed quantity y_i, and the standard deviation $\sum_i$ of y_i. The problem is then to determine a set of values for the variables $\mathbf{x}=[x_1,\cdots,x_n]$ that minimizes the χ^2 statistic.

We have used the performance measurement and analysis tool on several example programs on our Sun-3 workstation. We consider that our method is superior to performance analysis that examines only the program code because it can cope with a wider variety of program structures and does not depend on an unvalidated model of the underlying hardware. It is superior to earlier methods involving program measurement because it uses the programmer's knowledge about timing behavior and offers a quantitative measure of statistical confidence in the validity of the performance model that it generates.

III. Performance Polymorphism

One approach to addressing the requirement of producing results of different quality is to implement multiple versions of a function that carries out a given computation. These versions all perform the same task, but differ only in the amount of time and resources they consume, the system configuration to which they are adapted, the precision of the results that they return, and similar performance criteria. The versions may be specialized for a particular machine architecture, a particular problem size, a particular optimization strategy, and so on. The multiple versions may be supplied by the programmer, as when different algorithms adapted to problems of different sizes are supplied. They may also be generated automatically, as when a program rewriting tool is used to adapt a sequential program to a vector or parallel machine.

Given the multiple versions and the performance requirement for an instance of the real-time application, the compiler system then chooses the version which can produce the best result and still satisfy the performance requirement. We call this model *performance polymorphism*, because of its similarities to the conventional polymorphism where different functions carry out the same operation on different types of data. We have investigated the problem of binding in performance polymorphism, and the problem of how a system may allocate its available time and resources among performance-polymorphic subtasks.

The example we use to illustrate the model is that of sorting on a parallel computer system. We have (at least) three different sorting techniques available: a fast insertion sort *isort*, a heap sort *hsort*, and a parallel merge sort *bsort*. The amount of time that these sorts take is An^2+Bn+C, $Dn\log n+E$, and $F(n\log n)/p+G\log p+H$, respectively, where n is the number of elements to sort and p is the number of processing elements given to the parallel sort. We expect H, the overhead of starting the parallel sort, to be quite large. It may therefore be more effective to sort a short list of numbers on the local processor, without using the parallel sort. We also expect the constant factor in the time required for heap sort to be greater than that for insertion sort; it therefore may be more effective to choose the $O(n^2)$ insertion sort for extremely short lists.

To manage this information, a number of different items need to be taken into account. The programmer needs some way to specify the constraints on time and resources that are in effect, in order to determine when the choice of a version is feasible. In addition, a figure of merit for a given version may have to be defined, in order to choose the better of two feasible versions. The *binding* problem, the problem of determining the most appropriate version, must then be solved. Solving the binding problem demands information about the expected performance of versions of the program, and the system must therefore provide a capability of predicting the performance of a version. Finally, the situation where a single set of constraints governs the selection of different versions of several functions may arise. In this situation, we must solve the allocation problem, the problem of determining the most effective division of resources among the different functions.

We have implemented the system described above, and are using it to build trial systems and gather further information about its performance. Despite the fact that we do run-time binding only, we find that for medium to coarse-grained problems, the overhead of the binding of performance polymorphic

functions is negligible, adding perhaps a few hundred microseconds to the execution time of a function that takes from hundreds of milliseconds to seconds. We find that using performance polymorphism for smaller functions is ill advised, but have found few if any functions that are naturally substituted at that fine a level.

IV. Scheduling Parallelizable Imprecise Computations

Recent advances in parallel processing technology improve the execution speed of real-time computations and increase the likelihood that real-time applications will meet their timing requirements. A real-time application usually is composed of real-time tasks each of which has a ready time and a deadline. To avoid timing faults, real-time system designers can reduce the execution times of real-time tasks by utilizing efficient parallel algorithms designed for a variety of powerful and cost-effective multiprocessors.

For systems with occasional overloads, it may not be always possible to meet the timing requirements of all real-time tasks. One possible approach to avoid timing faults is to use the *imprecise computation* model. In real-time applications that support imprecise computation, every task is logically decomposed into a hard task and a soft task. We are interested in finding schedules which meet the timing constraints of all parallelizable tasks in a real-time application that supports imprecise computation. A schedule is *optimal* when the weighted sum of all unexecuted portions of the soft tasks is minimized.

We have studied an approach for finding schedules for the *imprecise multiprocessor scheduling* problem. More specifically, the problem of time allocation for independent parallelizable imprecise computations is formulated and solved as a linear programming problem. These parallelizable tasks, depending on how they are decomposed and scheduled, create different multiprocessing overheads. We characterize these multiprocessing overheads by linear functions of the degree of parallelism. Efficient algorithms designed for linear programming are used to solve the problem. If multiprocessor overheads can be accurately represented by linear models, then the time allocation is *optimal* in the sense that each hard task is completed before its deadline, and the weighted sum of the unexecuted portions of the soft tasks is minimized.

The linear programming solution decides only how much time should be allocated to each task, but without specifying which of the processors should be used. The problem of processor assignment given the time allocated is another difficult NP-complete problem. Currently, we use a heuristic algorithm for constructing a preemptive multiprocessor schedule from the linear programming solution. The algorithm first finds a good partition of the subtasks to be executed on each processor at each time segment. It then finds a sequence of time segments on each processor in order to minimize the number of task preemptions needed. We have used the algorithm to solve some medium size problems. on multiprocessor machines.

PI Names: Kwei-Jay Lin and Jane W.S. Liu

Institution: Univ. of Illinois, Dept. of Computer Science

Phone Number: (217) 333-1424

E-mail Address: {klin, janeliu}@cs.uiuc.edu

Contract Title: Utilizing Imprecise Results in Real-Time Systems

Contract Number: N00014-87-K-0827

Reporting Period: 1 Sep 87 - 31 Aug 90

Lists of Publications, Presentations and Reports

1. Journal Papers:

(1) Chung, Jen-Yao, J. W. S. Liu, and K. J. Lin. "Scheduling Periodic Jobs Using Imprecise Results," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1156-1174, September 1990.

(2) Lin, K. J. and S. Natarajan. "FLEX: Towards Flexible Real-Time Programs," *to appear in the Journal of Computer Languages*, Pergamon Press, New York, 1990.

(3) Chen, M., and K. J. Lin, "Dynamic priority ceiling protocols," *to appear in the Real-Time Systems Journal*, Kluwer Academic Publishers, Boston, 1990.

2. Conference Presentations:

(1) Chen, M., J. Chung and K. J. Lin, "Scheduling algorithms for coalesced jobs in real-time systems," *Proc. 13th IEEE Computer Software & Application Conference (COMPSAC)*, Orlando, FL, pp. 143-150, Sep. 1989.

(2) Han, C., and K. J. Lin, "Scheduling parallelizable jobs on multiprocessors," *Proc. 10th IEEE Real-Time Systems Symp.*, Santa Monica, CA, pp. 59-67, Dec. 1989.

(3) Lin, K.J., and C. Han, "Scheduling imprecise computations on parallel real-time systems," *Proc. 4th IEEE Annual Parallel Processing Symposium*, Fullerton, CA, pp. 62-76, April 1990.

(4) Lin, K.J., J. Chung and J. Liu, "Scheduling real-time computations on hypercubes with load balancing," to appear *Proc. 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990, (to be published as a book in 1991).

(5) Chung J.Y., J. Liu and K.J. Lin, "Optimistic token-driven reliable sequenced broadcast protocols," *Proc. 1990 Int. Conf. on Parallel Processing* (ICPP), Chicago, IL, pp. III303 - 310, Pennsylvania State University Press, Aug. 1990.

(6) Kenny, K.B., and K.J. Lin, "Implementing real-time systems using performance polymorphism," to appear *Proc. 14th IEEE Computer Software & Application Conference (COMPSAC)*, Chicago, IL, Oct. 1990.

(7) Gregory, G.J., and K.J. Lin, "Building real-time imprecise computations in Ada," to appear *Proc. TriAda '90*, Baltimore, MD, ACM, Dec. 1990.

(8) Kenny, K.B., and K.J. Lin, "Structuring large real-time systems with performance polymorphism," to appear *Proc. 11th IEEE Real-Time Systems Symp.*, Orlando, FL, Dec. 1990.

(9) Kenny, K.B., and K.J. Lin, "A measurement-based performance analyzer for real-time programs," to appear *Proc. IEEE Int. Phoenix Conf. on Computer and Communications*, Phoenix, AZ, Mar. 1991.

3. Technical Reports:

(1) Kenny, K.B., and K.J. Lin, "Implementing timing constraints in FLEX," Technical Report UIUCDCS-R-90-1567, Dept. of Computer Science, UIUC, Jan. 1990.

(2) Kenny, K.B., and K.J. Lin, "A measurement-based performance analyzer for real-time programs," Technical Report UIUCDCS-R-90-1606, Dept. of Computer Science, UIUC, Jun. 1990.

PI Names: Kwei-Jay Lin and Jane W.S. Liu
Institution: Univ. of Illinois, Dept. of Computer Science
Phone Number: (217) 333-1424
E-mail Address: {klin, janeliu}@cs.uiuc.edu
Contract Title: Utilizing Imprecise Results in Real-Time Systems
Contract Number: N00014-87-K-0827
Reporting Period: 1 Sep 87 - 31 Aug 90

Transitions and DoD Interactions

In the past year, several invited lectures were given at various universities and corporations including the Univeristy of Virginia (Jane W.S. Liu), NEC (Kwei-Jay Lin), and Fujitsu (Kwei-Jay Lin). Professor Lin has also given a presentation at a workshop on Fault-tolerant Real-time Systems jointly sponsored by ACM and ONR in Seattle, WA, in May 1990.

PI Names: Kwei-Jay Lin and Jane W.S. Liu
Institution: Univ. of Illinois, Dept. of Computer Science
Phone Number: (217) 333-1424
E-mail Address: {klin, janeliu}@cs.uiuc.edu
Contract Title: Utilizing Imprecise Results in Real-Time Systems
Contract Number: N00014-87-K-0827
Reporting Period: 1 Sep 87 - 31 Aug 90

Software and Hardware Prototypes

We are currently implementing several software prototypes including Flex compiler, Real-Time Mentat Language System, and Performance Measurement Environment. These software will be made available in the future.