Implementing real-time systems using performance polymorphism

Kevin B. Kenny Kwei-Jay Lin
Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801

Abstract
A new model for complex real-time systems is proposed. In this model we provide several versions of a program fragment to perform a particular action. These versions will differ only in their performance parameters, such as the time required, the resources consumed, and the precision of the results. This paper describes an implementation of a technique called performance polymorphism, in which the process of selecting a version from this set may be automated.

1 Introduction
A hard real time system is defined to be a system that must carry out some set of computations, each of which must be completed by a specified deadline. Frequently, failure to meet a deadline is intolerable, as the results of the computation are critical to the safety of life or property. Traditionally, the design philosophy of real-time systems has been to make the systems such that one may meet deadlines by choosing implementations of the algorithms that are faster, but inferior in some other respect such as resource consumption or precision.

The central problem in running a real-time system, then, is determining a way to perform all the required computations in order that all deadlines are met. The computations will have resource constraints: each computation will require certain resources, such as processors, memory, I/O devices, and communication channels, and many of these resources cannot be used concurrently by multiple computations. We must therefore find a schedule that defines when to start and stop computations in order to satisfy both the deadline and resource constraints. Scheduling is both facilitated and complicated by the fact that alternative versions of a computation may be available. The alternatives may provide additional flexibility, by allowing the scheduler to exploit tradeoffs between time and other resources, while at the same time making the performance of the overall system more difficult to characterize. These alternatives may be provided either by the programmer or by the programming system. For instance, vectorizers and parallel-processing compilers may produce versions of a program that are adapted to a parallel or vector architecture; the original serial problem may perform better in cases where, for instance, communication bandwidth is a critical resource constraint. In this paper, we present an implementation of a technique called performance polymorphism. Under performance polymorphism, whose name is chosen by analogy with conventional data type polymorphism, there can be multiple versions of a software component, each of which performs the same operation but meets different performance constraints. All of the versions are made available automatically, and the system chooses a version based on the constraints in effect.

Section 2 of this paper describes our model for performance polymorphism. Section 3 shows how essential performance data are derived. Section 4 describes the language support needed, and Section 5 discusses the issues of implementing an actual system that supports our model. Finally, Section 6 presents our conclusions. The remainder of this section describes the programming system on which our implementation of performance polymorphism is based.

1.1 The FLEX language
The Concord project at the University of Illinois has been working with an experimental programming language for real-time systems, called FLEX. FLEX implements two fundamental concepts that are missing from traditional programming languages: imprecise computations and constraint blocks. Lin, Natarajan, and Liu have developed the model of imprecise computations to deal with time-constrained iterative calculations. Imprecise computations represent the possible ways in which a computation may be terminated in order to return less precise results but take less time. This tradeoff is partly motivated by the observation that many real-time computations are iterative in nature, solving a numeric problem by successive approximations. Terminating an iteration early can return useful imprecise results. Imprecise computations are one source of alternative versions of a function.

Constraint blocks are the means by which FLEX describes the time and resources that are available for a particular computation. Lin and Natarajan have presented the mechanism of expressing timing and resource requirements using constraint blocks. A constraint block consists of a Boolean expression, followed by a block of statements during whose execution the expression must remain true:

constraint-block:
  opt-label ( expression ) { statements }

In addition to the program variables, there are several named items that may appear in the expression, representing

---

1This research was sponsored by the National Aeronautics and Space Administration under grant NASA-NAG-1-613, and by the Office of Naval Research under grant N00014-89-J-1181.
timing constraints:

- **start.** The absolute time at which the execution of the constraint block begins.
- **finish.** The absolute time at which the execution of the constraint block ends.
- **duration.** The relative time during which the constraint block is in execution. It is the time elapsed from start to finish.
- **interval.** The relative time between successive executions of the constraint block. It is the time elapsed from the previous start to the current start.

Some examples of the constraint expression are as follows:

**Example 1 (finish < x)**
The finish time of the statements must be less than absolute time x.

**Example 2 (interval == 30; radiation <= 10)**
The block must be executed at intervals of 30 seconds. Attempting to start too soon will delay execution until the correct start time; attempting to start too late will cause a timing fault. A fault will also be detected if, at any time during the block's execution, the value of the variable radiation exceeds 10.

2 A model for performance polymorphism

Let a 'PPF' denote a performance polymorphic function, i.e., a function F for which some set of v versions \{F_1, F_2, ..., F_v\} is provided, each of which carries out function F, but which have different performance criteria. Each of these versions will use some amount of time and resources, possibly depending on the data presented to it. Each version will also have a figure of merit that determines how attractive the use of that function is; a common figure of merit might be the amount of CPU time consumed, or the precision of the available results. Without loss of generality, we assume that a larger figure of merit represents a more desirable choice; we take the arithmetic inverse of the figure if necessary to impose the correct ordering.

The binding problem for PPFs is the problem of determining which of the candidate F_i’s to choose for an invocation of a PPF F. We must also decide when to make that determination (which may be as early as compile time or as late as the actual function call).

The binding process, rather than consisting of a single step, may be a process of gradual elimination of choices as information becomes available about the resources available for the function invocation. There are two major reasons why a candidate F_i may be eliminated. The first is that F_i is infeasible, i.e., we know that there is no possible configuration of resources (given our incomplete knowledge about resource availability) for which all F_i’s constraints can be met. The second is that F_i is covered by another function F_j, i.e., wherever F_i is feasible, F_j is also feasible, and moreover a higher figure of merit is achievable using F_j than using F_i.

We can make the choice of a particular F_i as soon as we can determine that F_i is feasible everywhere and covers all other candidates F_j (j ≠ i). This condition can be reduced to a Boolean predicate, can-bind(F_i), at compile time. The invocation of a PPF F becomes a set of guarded commands:

\[
\begin{align*}
\text{if} & \quad \text{can-bind}(F_1) \rightarrow F_1(x_1, x_2, ..., x_n) \\
& \quad \text{can-bind}(F_2) \rightarrow F_2(x_1, x_2, ..., x_n) \\
& \quad \vdots \\
& \quad \text{can-bind}(F_v) \rightarrow F_v(x_1, x_2, ..., x_n) \\
\text{fi}
\end{align*}
\]

The choice of binding time can then be treated as a data flow analysis problem, in an analogous manner to the choice of binding time in type polymorphism\cite{5, 6, 7}. The guards are Boolean expressions with a good deal of redundancy, and we perform operations such as constant folding, common subexpression elimination, and code motion to simplify the guards.

For a concrete example, let us consider a PPF F that sorts a list of objects. We know t, the amount of time that is available to perform the sort. We also have information about n, the length of the list to be sorted. We have two candidate functions, F_i, an insertion sort, and F_a, a heap sort (or other \(O(n \log n)\) sort). We know the amount of time that each of these candidate functions may take: they are \(A_n^2\) and \(B n \log n\) respectively. (A and B are constants that are determined either by analysis of the code or by measurement.) We want to complete the sort in the minimum possible time. The set of guarded commands for this function invocation would then look like:

\[
\begin{align*}
\text{if} & \quad A_n^2 \leq t \land A_n^2 \leq B n \log n \rightarrow F_i(...) \\
& \quad B n \log n \leq t \land B n \log n \leq A_n^2 \rightarrow F_a(...) \\
\text{else} & \quad \text{timing fault.} \\
\text{fi}
\end{align*}
\]

The first part of each guard is the test that the function satisfies the deadline constraint; the second part is the test that the function has the highest figure of merit among the functions that meet the deadline. A few obvious redundancies have been removed; it is obvious how the common subexpressions could be eliminated and the guards simplified.

3 Deriving performance information

The information required for the performance binding decision falls into two general classes:

- **environmental** information, i.e., information regarding the environment in which the PPF is invoked. The environment, for our purposes, comprises the constraints on time and resources, and the values of the uncontrolled variables. Environmental information propagates inward; i.e., it is provided by the caller of a PPF, and exists in a more global context.

- **performance** information, which describes the performance of the PPF given its environment. Performance information propagates outward, i.e., it is provided locally for a function, and is used in a more global context to influence the binding decisions.
Any implementation must allow the user to specify both the environmental and performance information. The environmental information is the more straightforward of these; the constraints can be expressed using constraint blocks in FLEX. For example, an invocation of our sort function, and its associated constraint block, might look like:

```
(duractioo <= T && heap space <= U) {
    sort ( ... )
}
```

This constraint block states that the sort must complete in at most $T$ seconds of elapsed time, and use no more than $U$ units of heap space.

The performance information, however, is more difficult to obtain. There are two fundamental approaches to predicting it; we call these approaches analytical and empirical.

The analytical approach is the more customary of the two, being used in the systems of Stoyenko[8, 9], Mok[10], and Puschner[11]. In this approach, the time required to execute basic blocks of a program is calculated by examination of the machine code. The times of the higher-level constructs are then derived by combining the basic block times with information regarding the number of times that the basic blocks are executed.

While they are attractive, these systems suffer from several drawbacks. They depend on a characterization of the performance of the target machine. Issues such as caches, paging systems, and bus contention, can make the performance difficult to characterize. More importantly, they are attempting to decide an undecidable problem, as calculating the number of times that a basic block is executed is equivalent to solving the halting problem. They avoid the undecidable situation by restricting the programmer to a limited set of constructs, forbidding unbounded loops, recursion, and similar constructs with open-ended timing behavior. Moreover, they cannot take into account the user's knowledge about the performance behavior of functions.

The empirical approach, on the other hand, uses measurements of the actual timing behavior of the code, and attempts to fit these measurements to a parametric model supplied by the programmer[12]. This approach ensures that the performance figures are based on the actual hardware. There is no uncertainty introduced by a model of the machine's timing behavior. There is no attempt to solve an undecidable problem, and therefore there need be no restrictions on the language. A measure of statistical confidence is provided, and the system designed can use this to validate the model parameters.

On the other hand, the empirical approach does not guarantee worst-case performance. It may state that the worst case ever observed of the function's performance meets the criteria, but it cannot ever guarantee that a worse case will not arrive in the future. Moreover, this sort of end-to-end analysis is inappropriate in analyzing the behavior of systems with multiple interacting components, where the performance may be determined by unrelated parts of the system; an example is that the time delay needed to enter a critical section does not depend on the particular invocation, but rather on load factors elsewhere in the system.

Our system uses a synthesis of these two techniques. We analyze the lower-level components empirically, and then treat the result (appropriately augmented with safety factors) as if it came from an analytic model. The analytic approach may then be used to characterize the performance

```
void sort (void *array, // Array to sort
         int n, // Count of elements
         int size, // Size of an element
         COMPAR *cmp) // Comparator
    perf_poly; // The sort is
                // performance-polymorphic
```

Figure 1: A performance-polymorphic function

of higher-level chunks, and to estimate such quantities as critical section and synchronization delays.

4 Language support

To implement performance polymorphism, we require several new features in the implementation language. It must have a syntax for declaring performance polymorphic functions. A means must be provided to describe the figure of merit of a version, so the system can make a choice from among multiple feasible versions. Finally, when a function is invoked, the appropriate actions have to be taken to bind a candidate function to the invocation. In this section, we consider these issues.

4.1 Performance polymorphic functions

In order to support the empirical evaluation of software components, and to add performance polymorphism, we must make some additions to the FLEX language.

Performance polymorphism is the easy one of these; we simply need some way of declaring a PPF. To do this, we augment the FLEX language with

- a means of declaring a PPF $F$, and
- a means of declaring that a specific function is an implementation of $F$.

The first of these is quite simple: it is an extension to the syntax of the "function body" of C++:

```
fct-body:
    perf_poly
```

The second is a new style of declaration in the language:

```
declaration:
    provide-declaration
```

```
provide-declaration:
    provide identifier, for identifier;
```

This declaration indicates that the function $identifier_1$ is an implementation of the PPF $identifier_2$, which must be declared as $perf_poly$ in the current scope.

To study this, let us examine the declaration of several different kinds of sort function. Assume that each function accepts four parameters: the address of the start of an array, a count of elements, the size of each element, and the address
#include "sort.h"

// Declare that we're providing an implementation
// of the performance polymorphic function "sort"
provide my_qsort for sort;

// Implementation of "sort" that calls "qsort".
void my_qsort (void *array, int n, int size, COMPAR *comp)
{
    // Fit a quadratic to the length of time
    // needed for short lists; fit a (n log n)
    // function for longer ones.
    #pragma measure worst cpu_time \ defining a, b, c, d, e \ in (n < 30) ? \ (a * n + b) * n + c \ : d * n * log (n) + e
    { // Call the library function
        qsort (array, n, size, comp);
    }
}

Figure 2: Instrumenting the qsort function.

4.2 Figures of merit

Once we have the environmental and the performance information characterized, we are nearly ready to determine the binding for any PPF invocation. The environmental information specifies the resource constraints; the performance information specified the resource claim. We still need to specify the figure of merit; it must be specified at the PPF invocation, as different invocations may have different objectives. We specify it with another #pragma directive:

merit-directive:

#pragma objective merit-objective expression

merit-objective:

maximize | minimize

In the example from Section 3, we would augment the constraint block by giving a directive that, in addition to meeting the deadline, the objective is to minimize the amount of real time consumed:

(duration < T && heap_space < V) { #pragma objective minimize duration sort { ... }
}

Given all of these sources of information, the compiler can construct the guards with the can-bind predicates, as shown in Section 2. It can then perform data flow analysis to reduce the cost of evaluating the guards, by partially evaluating them at compile time, and hoisting them out of inner sections of code.

5 Implementation issues

5.1 The programming environment

The global design of our system is shown in Figure 3. The compiler accepts FLEX programs, augmented with PPFs and measurement directives. It produces object code for the FLEX program, including probe points to perform the required timing measurements.

During program initialization, the program reads a data file that is created by a previous run of the program analyzer. It contains the values of fit parameters in the parametric model that give the best fit for the measured data. These parameters are stored in external variables in case the user’s scheduling system requires them; they are also used to calculate the expected performance when attempting to bind PPFs.

As part of its normal operation, the program writes out a data file with observed performance numbers from the #pragma measure directives. These numbers provide input to the program analyzer. The data in this file are raw; they do not depend on the fit parameters and thus may be preserved from run to run.

The dynamic analyzer’s job is to compute the amount of time and resources required for sections of the program to carry out their functions. This information is used not only for performance polymorphism, but also to verify the conditions on constraint blocks. The analyzer updates the fit parameters for all the measured statements. The updated parameters are written to the parameter file so that the next run of the program can see them at initialization time.

Once the parameters of the performance model are available, we can perform the analyses to validate constraint blocks, and calculate the time commitments for PPFs. This determination is carried out by the static analyzer. It works
in much the same fashion as the analytical performance predictors described in Section 3.

The static analyzer will operate by propagating timing data in both a global-to-local (top-down) and local-to-global (bottom-up) fashion. In the global-to-local direction, the timing constraints on constraint blocks are viewed as limits that must be verified. All the component operations of a block must complete in a short enough time that the overall constraint is satisfied. In the local-to-global direction, the constraints are viewed as assertions; since the analyzer will verify each of the constraints individually, larger program components that are fabricated from the constraint blocks may assume that the assertions are valid. Other assertions are provided by the observed worst-case behavior of measured segments of the program; if we cannot predict the performance of a section of code a priori, we can measure it and use the measured performance as an assertion regarding the expected performance, adding an appropriate safety factor.

If instruction-level timing analysis is available, it can also enter into the static analysis. Basic block times can be calculated directly by the instruction-level analyzer (which must have information available about the correspondence between basic blocks at the source level and the machine instructions); these times can be combined into the times of larger units, either by formal analysis of the control structure or by empirical measurement of the number of times that a lower-level block is executed (the count variable).

5.2 Binding PPFs
We have seen in Section 2 that an invocation of a performance polymorphic function is represented as a set of guarded commands. This representation is analogous to late binding in type polymorphism; the guards in type polymorphism would relate to compatibilities among the data types rather than to meeting performance specifications. This correspondence raises the issue of whether the performance of the PPF mechanism may be improved by adopting compile-time binding.

We prefer, however, to view the binding process as a continuum. The guards are Boolean expressions that depend on program variables; some of these variables represent the performance of a candidate function, while others represent the environmental information. These variables acquire values at various times—when input data become available, when deadlines are set, when performance measurements are made, and so on.

Given some incomplete set of values for the variables, we can eliminate some of the guards (because the corresponding functions are infeasible, or known to be inferior to other candidates), and simplify the evaluation of others. This elimination and simplification is analogous to the process of type inference in conventional polymorphism[5, 6, 7], and is therefore a data flow analysis problem. In fact, with the guards being Boolean expressions over the program variables, it is exactly the problem of global common subexpression elimination, and is addressed in virtually all modern optimizing compilers. We note that the variables used in the guards typically do not suffer from aliasing; the flow analysis will therefore be able to optimize quite aggressively.

Another related issue is the granularity of PPFs. There is no point in adding performance polymorphism to a system if the overhead incurred by the mechanism is greater than the performance to be gained. We see, though, that even with late binding, the binding decision can be reduced to a reasonably cheap and simple set of function evaluations. Fairly fine-grained procedures, like single steps in a differential equation solver, can therefore be made into PPFs.

5.3 Allocation and binding of multiple PPFs
So far, we have been assuming that we have a single PPF $F$ which has to use no more than a given set of resources. In practice, however, programs are more complicated; within a single set of constraints, we may have to perform tasks involving a set of PPFs, e.g., $F$, $G$, $H$. In addition to having each function meet its constraints, we must combine the amounts of resources needed by $F$, $G$, and $H$ to find the resource usage for the higher-level task. We must also optimize the task as a whole; to do this, we must know how to combine the figures of merit of $F$, $G$, and $H$.

The combination of these resource claims and merits is generally more complex than simple addition. For example, if $C_{F1}$ is the amount of working storage required for $F$, and $C_{G1}$ is the amount for $G$, the combination will not be $C_{F1} + C_{G1}$, but rather max($C_{F1}$, $C_{G1}$). For some applications, particularly for properties such as precision, the compiler may not know how to combine resources. The combination function may even differ from one application to another, despite having nearly identical program structures; for instance, one application may wish to optimize the mean bandwidth required on a communication channel, while another wishes to optimize the peak bandwidth.

Figure 3: A system with performance polymorphism
We have identified some major classes of resources and the combination functions that model them. The major resource classes can be described as time-like and space-like. A time-like resource generally models some sort of time: real time, CPU time, I/O channel time, and so on. A space-like resource models some physical object: dynamically allocated storage, processing elements, disc space, communication bandwidth. The functions for combining the commitments of these resources are shown in Table 1. These combinations and their implications are discussed more fully in [12].

Even once the combination functions are known, the problem of optimizing the larger set is likely to be computationally intractable. For instance, the problem of packing several tasks, each requiring a variable amount of time, into a fixed-length time until deadline is the NP-complete 'knapsack' problem. We therefore expect that the division of time between PPFs will be application dependent, and may be specified at each level of the program. One requirement of a high-level task is to determine a division of its resources among the low-level tasks (generally using some heuristic to attempt to optimize the resource usage). Once this division has been determined, the binding of the PPFs can take place.

One problem in resource distribution for which a class of efficient solutions is known is the 'knapsack sharing problem.' In this problem, several tasks have a tradeoff between time and one other resource, and the optimal division of the time among the tasks can be computed with little computational effort. Brown [13] presents algorithms for its efficient solution, which are beyond the scope of this paper. We foresee algorithms of this type having an important place in any scheme involving multiple performance polymorphic functions.

6 Conclusions

In addressing the problems of hard real-time systems, we often come upon problems where we must decide among qualitatively different actions based on quantitative information about resource constraints. Multiple versions of a function, having a homogeneous interface but meeting different constraints, may be present. Performance polymorphism is a unified theory to express the choice among these multiple versions in a way that is both natural and powerful. It allows the flexibility of adding new versions at any time, of adapting to unforeseen constraints, and of adapting to automatically generated variants of a procedure (as, for example, might come from a parallelizing compiler).

We have developed a means to implement the theory of performance polymorphism that requires very low overheads at run time, by analyzing the constraints and propagating performance information backwards in order to bind performance polymorphic functions as early as possible. We have identified the tools required for the implementation; all of them appear to be quite feasible with present software technology.

While there are, of course, other ways to view the problem of variant functions under performance constraints, we are confident that our method is feasible, powerful, and natural for the programmer.

References