Temporal Analysis for Adapting Concurrent Applications to Embedded Systems

Sibin Mohan
Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, USA
smohan@cs.ncsu.edu

Johannes Helander
Microsoft Innovation Center, Europe
Ritterstrasse 23, 52072 Aachen, Germany
jvh@microsoft.com

Abstract

Embedded services and applications that interact with the real world often, over time, need to run on different kinds of hardware (low-cost microcontrollers to powerful multicore processors). It is difficult to write one program that would work reliably on such a wide range of devices. This is especially true when the application must be temporally predictable and robust, which is usually be the case since the physical world works in real-time. Thus, any application interacting with such a system, must also work in real-time.

In this paper we introduce a representation of the temporal behavior of distributed real-time applications as colored graphs that capture the timing of temporally continuous sections of execution and dependencies between them, creating a partial order. We then introduce a method of extracting the graph from existing applications using a combination of analysis techniques. Once the graph has been created, we introduce a number of graph transformations that extract "meaning" from the graph. The knowledge thus gained, can be utilized for scheduling and for adjusting the level of parallelism suitable to the specific hardware, for identifying hot spots, false parallelism, or even candidates for additional concurrency.

The importance of these contributions is evident when we see that such graphs can be sequentialized to our partiture model and can then be used as input for offline, online, or even distributed real-time scheduling. Finally we present results from analysis of a complete TCP/IP stack in addition to smaller test applications which show that our use of different analysis models result in a reduction of the complexities of graphs. An important outcome is that increasing the expression of concurrency can reduce the level of parallelism required, saving memory on deeply embedded platforms, while keeping the program parallelizable whenever complete serializability is not required. We also show that applications which were previously considered to be too complex for characterization of their worst-case behavior are now analyzable due to the combination of analysis techniques that we utilize.

1. Introduction

Software applications that interact with the real world and devices or computers on the Internet are at the center of the next generation of consumer electronics. We are increasingly depending on such systems in our everyday lives in health care, robotics, infrastructure, entertainment and even clothing. Such applications and other cyber-physical systems run on different kinds of embedded hardware ranging from 8-bit microcontrollers to sophisticated multicores. Since we depend on these devices, they must be robust and as they interact with the real world, they must work in real time. Hence, their temporal behavior must be robust and predictable. Unfortunately, it is quite difficult to write one program that would work reliably on such a wide range of devices. This is particularly difficult if timing depends not only on the application, but also on hardware details and other applications on the device. This makes application development slow and expensive.

It is also important to pay attention to the hardware capabilities of a given target platform, such as the number of processors and amount of memory available. In a low-cost microcontroller the scarcest resource is often memory, particularly RAM. Our early experience with service-oriented cyber-physical systems [13] shows that one of biggest RAM users tend to be thread stacks, requiring a reduction in the number of threads. This implies that the parallelism needs to be reduced – *i.e.* the application needs to be executed sequentially. In contrast, on a high-end multicore processor the bottleneck is not memory but the amount of parallelism available in the application. It often makes sense to execute the same service application on both ends of the embedded spectrum with only the throughput and the number of services being varied. Thus, the applications need to be tuned in such a way as to address the bottlenecks on each platform – the application needs to be sequentialized for the low-end platform and parallelized for the high-end platform. The *future*, a delayed function call, originally proposed for handling synchronization in MultiLisp [11] and later used in other languages, is a way of expressing light-weight *potential* parallelism.

Having a model that would *(a)* enable analysis of the program's temporal behavior and *(b)* match it to a given hardware would be most helpful. In previous work [15], we showed how new programs could be written together with such a model. The model can be represented and manipulated as a graph, as discussed in this paper, or serialized to XML. We call the serialized version of the model a *partiture*, an expression analogous to a short score in music, where the conductor can see what instruments should play at a given time without regard to the details of how they do

1068-3070/08 $25.00 © 2008 IEEE
DOI 10.1109/ECRTS.2008.38

it.

In practice though, most pre-existing applications do not follow model-based development practices, but it is still desirable to adapt them to new platforms. This even applies to large software projects in general, where few, if any, engineers understand how a program actually behaves due to large development teams and changes over product revisions. Helping the engineers understand and modify complex software would be useful. We propose an automated process whereby we could transform applications into using *partitures* and *futures* [15], making model-based scaling possible, while keeping program execution correct. This would reduce the tedious and error prone methods for transforming applications by hand when they need to be deployed on new platforms. Such a tool, where we extract temporal models from existing applications, is introduced in this paper. The knowledge gained, can help designers of such systems in*(a)* optimizing programs for various platforms; *(b)* distributed orchestration, *(c)* adaptation and scheduling [15, 24] or *(d)* even executed on modeling engines [17] to check for correctness.

Note that multiple graphs can be merged, allowing analysis of several applications at the same time. Multiple applications can also be analyzed separately and properties of their compound behavior, such as minimum number of threads or schedulability, can be examined together. Handwritten partiture fragments can also be merged with automatically generated information, either in the graph form or as XML, thus providing a mechanism for inserting domain knowledge into the graph.

1.1. Static and Dynamic Analysis

Knowledge of the temporal behavior of an application is hidden inside the application logic, where it is extremely difficult to analyze and model for any given hardware. While static [5, 19, 22] and dynamic [25] timing analyses are used to obtain the worst-case execution times (WCETs) for real-time applications, they may not able to to provide a complete picture of a program. This is particularly true in the case of larger, more complex programs. Programs that contain function pointers are typically out of the reach of static analyzers. Dynamic analyzers are unable to gauge the true nature of the program and have shown to be unsafe [25] – *i.e.*, they may underestimate the WCET of the program, which could lead to dangerous effects. If the application uses concurrency constructs such as signals, locks or mutexes, then neither of these techniques can fully analyze the application. In this paper we study the use of combinations of variety of techniques to form the complete picture of the structure and execution characteristics of a distributed embedded application. The results obtained are collected to create *timing graphs*. The timing graph and the *partiture* are two representations of the same information. A *bar* corresponds to a node in the graph and triggers and sequences between the bars correspond to edges. This means that the graph can simply be converted to a partiture and then be used for purposes such as inputs for offline, online, or distributed real-time scheduling or converted to a model program [15].

1.2. Contributions

The main contributions of this paper are:
(a) Extend the scope of static timing analysis to more complex applications by combining it with other techniques, in particular run-time traces and type inference. Applications which were previously considered to be "un-analyzable" due to their inherent complexity, are now analyzed using our graph capture and transformation techniques so that their worst-case behavior can be characterized.
(b) Define a colored graph representation of a program's temporal behavior, including invariants and transformations. The graph corresponds to a *partiture*, a programmatic expression and transitively, to a model program that can be executed on a modeling tool.
(c) Derive information from the topology of graphs, allowing an engineer to optimize an application such as to make it more scalable and amenable to being transformed into our application model (partitures,futures,*etc.*). Some of the knowledge that can be gleaned is the minimum number of threads required for an application to correctly execute, graph sections with potential false parallelism, and dependencies that prevent parallelization.
(d) Observe that adding concurrency can save memory and present a method to an engineer for pinpointing areas where concurrency can be increased.
(e) Due to the incremental nature of the analysis, even an incompletely understood application can be explored, allowing an engineer to learn something about the application behavior and augment the automatically generated model with manually provided domain knowledge. This is a critical step forward as embedded designers now have more choice in the type of applications that they can develop, especially for time-constrained systems.
(f) Demonstrate that creating the timing graph and performing subsequent transformations is feasible by means of presenting an implementation that was applied to actual embedded software – the TCP/IP stack of an embedded operating system.

We believe that the use of the combination of analysis techniques presented in this paper, to enable the process of extracting temporal behavior of existing applications which ultimately leads to ease of development of distributed embedded applications on varied platforms, is, to the best of our knowledge, a first of its kind.

The rest of this paper is organized as follows: we discuss the use of *futures* in embedded software in section 2. The colored graph representing a program's temporal behavior is introduced in section 3, together with invariants that define a valid graph. The algorithm for creating temporal graphs is introduced in section 4. Transformations that can be used to simplify and/or reveal interesting topological properties of the graph are defined in section 5. Section 6 explains the insights obtained from the graph transformations. Section 7 details the experimental framework

and methods to collect the raw data required for graph generation. Section 8 shows results, followed by related work and conclusion in sections 9 and 10 respectively.

2. Saving Memory through Sequential Execution

In low-end microcontrollers a multi-threaded application uses one stack for each thread. Since microcontrollers do not usually have a memory management unit, each thread stack must be allocated from physical memory. The maximum size of the stack is limited by the available memory, while the minimum is the largest stack the thread can ever need. Once a thread has been created it cannot give up its stack, whether it is running or blocked, since there are live stack frames occupying part of the stack. This is the case even before the thread has run for the first time as an initial stack frame must be created by the runtime. The stacks cannot be moved as frames in it contain pointers to data in other stack frames and cannot be compacted as executing code may need to push additional frames. As a final option copying, compressing, and decompressing stacks at each context switch to the side and sharing the actual stack between threads would be complex and inefficient. Clearly threads are problematic on low-end embedded systems.

The most common alternative is an event loop. Instead of creating a thread to handle a sensor reading, for instance, an event is posted. A loop in the program then picks up an event and examines it. The disadvantages to this are – *(1)* all applications turn into state machines with complex interactions – the code becomes hard to understand and *(2)* the development process becomes error prone with skyrocketing maintenance costs.

One approach is to transform an application written with threads to use the so-called split-phase operation [1] where the temporal phases of a thread are split into separate functions. However, no automated transformation from existing programs has been available however, meaning significant engineering effort is required for such a transformation. The approach has been codified in the NesC programming language [9]. Unfortunately the split-phase mode is essentially the same as the event driven model, including the need to communicate state from one phase to another in global variables or objects pointed to by global variables.

One attempt at combining the features of threads programming and event handling is protothreads [4], where stacks are unwound at blocking points. While this appears to work only in the C programming language, the implementation relies on non-standard features in a specific compiler (which it uses in a clever way). More importantly protothreads do not save local variables during blocks, making the appearance of thread programming at best an appearance and at worst an endless source of difficult bugs.

Futures [6] were originally proposed in the Lisp community as a way of deferring evaluation and increasing performance [8]. They were used as a primary construct for concurrency and synchronization in MultiLisp [11]. Futures have also been implemented in mid-level languages, such as Java [3] or C#. Futures have been natively implemented in C, on a microcontroller, in our earlier work [15]. When C programs are written in an object-oriented fashion, it is easy to turn any method call into a future with very little modification to the program. Threads can also be converted to futures. Creating a future is similar to calling a method or function, with the big difference that the call is executed asynchronously. Parameters are delivered as in a regular call. If a split-phase program would be rewritten in terms of futures, the various phases could send values to the following phase in normal function parameters, including *this* pointers in object-oriented programs, thus obviating the use of global variables, an engineering practice commonly advocated in the last several decades. Compared to protothreads, the normal language rules are in effect and asynchrony is explicit and controlled.

Instead of being implied or encoded in the program, such as in traditional threads programming,timing parameters and any required concurrent execution is expressed in a partiture, where each future is associated with one or more bars. The advantage of futures over threads is that futures can be inserted anywhere (there is *no predetermined parallelism, only concurrency*), stack allocation can be deferred until the future is ready to run and the cost of creating a creating a future is low. Control loops can be moved into the partiture leaving just the worker function in the future. Thus a typical construct where a thread waits for an event, then processes it and then waits again can, with little effort, be changed to give up its stack during the wait. This is why futures exhibit all the advantages of split-phase operation, when generously used. However, it is not always necessary to convert all blocking points to futures so as to make it possible to run the entire software on exactly *one* stack. Often an embedded device has room for a small number of stacks, such as two. Section 6 shows how to discover the number of stacks required and how to reduce the number one step at a time.

The real strength of futures is, however, in parallelizing the program. On a multicore processor futures can be executed in parallel. The parallelism is only limited by the dependencies between futures, which are conveyed to the scheduler in the partiture. The future, combined with *partitures* [15] and the "temporal timing analyzer" presented in this paper allows for an aided and incremental program transformation that has all the positive features of split-phase operation while being more flexible and structured. This allows programs to execute in parallel when parallelism is available on the hardware and sequentially otherwise. The future is an explicit expression of concurrency. Adding concurrency thus, not only adds *potential* parallelism, but also reduces the *required* parallelism, thus saving memory.

False parallelism between two (or more) threads refers to the situation where in reality such threads can actually only execute *sequentially*. Using threads with *false parallelism* between them leads to a requirement for multiple stacks. This is not really necessary but is merely an arti-

fact of the programming model. With futures, such dependencies can be broken or made explicit if *real* parallelism exists.

3. The Timing Graph

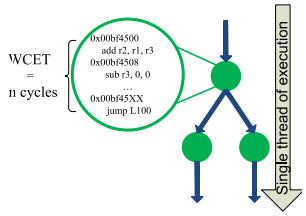
At the core of our analysis is the *timing graph*, which is a graph that enumerates the timing and execution characteristics of a program (including concurrent programs).

3.1. Representation of the Timing Graph

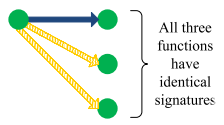
To reason about and distinguish between the various constructs in the timing graph in a precise manner, we allocated colors and corresponding shapes to the nodes as well as for each type of edge. The five colors used are depicted in Figure 1.

Code that runs within a single thread without external interactions is represented by *green (circular)* nodes (Figure 1(b)). These nodes represent straight line code, possibly entire functions or call graphs between temporal program phases. A temporal phase is delimited by potential sleeping, waiting, signaling, message passing, or other points of interaction with other threads (or futures). Green (circular) nodes correspond to *bars* in the *partiture*. Regular timing analysis is performed to obtain the WCET of this block of code. Transitions between the green nodes within the same thread of execution is represented using *blue (solid)* arrows. Figure 1(c) shows *yellow (dotted)* edges, which are "possible" blue edges, where it is not known whether the program could ever make the transition or even if it could not.

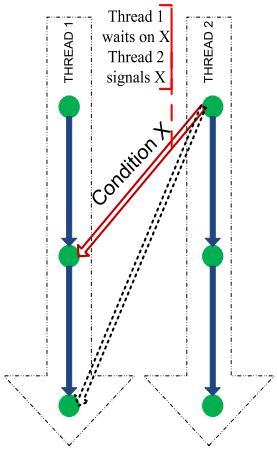
Applications consisting of multiple threads that communicate by use of various concurrency constructs are illustrated in Figure 1(a). Blue edges are restricted to their own threads. Calls to communication constructs are depicted by use of *red (hollow)* edges, where an incoming edge represents a *wait* and an outgoing edge a *wakeup/signal* on a shared synchronization object. Further analysis reveals other possible synchronization objects that could be



(b) Green nodes and blue edges of the timing graph



(c) Yellow edges – possible blues



(a) Red/black edges – interactions between threads

Figure 1. Edges and Nodes in the Timing graph

condition_wait(x) = wait for condition 'x' to be signalled
condition_signal(y) = signal condition 'y'. Basically wake up the next thread waiting for this condition.

Figure 2. Synchronization constructs used in our framework

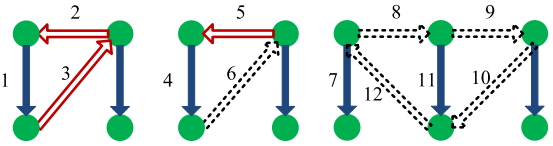
called/waited upon leading to more edges in the graph. These edges are colored *black (dotted hollow)*, representing "possible" reds. The black edges represent the signaling of an unknown object, such that the wait and the signal are the same object. If it turns out that the wait and signal are for different synchronization objects then the edge is eliminated. For our analysis, and for the sake of simplicity, we assume that all concurrency constructs are one of the two defined in Figure 2. All other basic synchronization constructs face situations where one process waits while the other signals – similar to a condition.

3.2. Graph Invariants

The timing graph has certain invariants that must never be violated, either during the creation process or while performing one of the transformations: *(a)* the final schedule that is created will retain all dependencies among the various threads and *(b)* a transformation must not, inadvertently, make the program incorrect. This second invariant is important – *i.e.*, if a transformation will result in the creation of a deadlock, then that transformation is not performed. In the case of the timing graph, a "deadlock" is defined as a directed cycle formed by one of the following:
(a) red edges only
(b) red edges with one or more black edges
(c) one or more blue edges with one or more red and/or black edges.

In Figure 3(a) edges "1", ""2 and "3" form a cycle, while in Figure 3(b) edges "4", "5" and "6" form a cycle – hence, these two graphs have deadlocks. Deadlocks formed *only* by black edges though (Figure 3(c)) are acceptable because black edges represent alternate schedules – the deadlocking black edges cannot exist in the same graph/schedule, since it is assumed the program is valid unless proven otherwise. This means that in an actual schedule we do not know ahead of time which condition variables will be used, but can guarantee that it will be only those that do not form a deadlock. Hence, our analysis can proceed in the presence of "false deadlocks" formed by black edges based on the premise that the program is, and will remain, valid.

For real-time programs, timing issues will also be a fac-



(a) Reds/Blue (b) Red/Black/Blue (c) Not a deadlock

Figure 3. Deadlocks in Timing Graphs

tor. A full real-time schedulability analysis will take these additional factors into account and perform constraint solving. However, the invariants mentioned above will still be true and the graph transformations that we present in this paper are just as valid for real-time schedulability. In such cases, the graph is evaluated with the above topological invariants in the first pass. Schedulability analysis can then be performed in the second pass, with an extended set of invariants, but we will not delve into the details as this is beyond the scope of this paper.

4. Information Sources for Graph Creation

In this section we shall enumerate the process of creating such a graph – *i.e.*, show how the information is gathered from a variety of sources and then put together to create the actual graph. Section 4.1 enumerates the various sources of information while section 4.2 shows how all of it fits together to form the graph.

4.1. Information Gathering Techniques

We use a variety of sources to gather the information that is used to create the timing graph. They are a mixture of static and dynamic data as well as higher level information.We use *four* such techniques, combined together, to obtain the complete picture of the control flow and dependency behavior of the application. The sample application in Figure 4 is used to the explain each step, while Figure 5 shows the final results obtained after applying these techniques.

(1) Static Analysis: The control flow graph (CFG) of the application is created and analyzed at compile time to obtain the static function call graph of the application. Static analysis shows us that function "foo" calls "func1" (Figure 4), represented by the horizontal blue arrow between the two nodes (Figure 5).

(2) Dynamic Tracing: We execute the program with sample inputs and trace the function calls during execution. We are able to find *some* of the dynamic calls that were made in the program, represented by calling "func2" using the function pointer "fptr" (Figure 4). This gives us a better picture of the control flow in the program and results in the addition of the vertical blue arrow in Figure 5.

```
void func1( int i ) ;

int func2( int i, double d ) ;
int func3( int i, double d ) ;

double func4( char c1, char c2, int i ) ;

void foo()
{
  void (*fptr)( int, double ) ;
  func1( 10 ) ; // static call
  fptr = func2 ;    (*fptr)( 10, 5.0 ) ; // dynamic call
}
```

Figure 4. Sample code to illustrate creation of the Timing Graph

(3) Type Information: Once step (2) is complete, we compare the type information of all other functions in the application to see which ones have the *potential* to be called. If the signature of a (dynamically) called function matches that of another, uncalled function, then there is a possibility that the latter may be called at the *same call site*. Functions "func2" and "func3" (Figure 4) have identical signatures. Since "func2" was called there is a possibility that the same function pointer could have called "func3" as represented by the yellow arrow (Figure 5). Type information can also be used to reduce the universe of possibilities for dynamic function calls. For instance, between the static analysis and dynamic tracing phases, there was a possibility that any one of the functions in the program (func1, func2, func3, func4, or even foo) could have potentially been called using the function pointer. Once dynamic analysis tells us that "func2" was called, we can immediately eliminate "foo", "func1" and "func4" from the list of possibilities because its type is different from that of "func2". The true value of type information comes when trying to gather information about concurrency constructs. If, during runtime tracing, we are able to gather the information that a particular concurrency call was made, we can limit the possibilities of other concurrency constructs that the call site can then invoke by the use of type information of the first callee.

(4) Incremental development with inputs from domain expert/programmer. We can further prune outgoing edges by inputs based on domain knowledge. For instance, we were able to gather that 'func3" is a potential callee because its type matches that of "func2". A domain expert might be able to point out that based on the application design there is *never* a possibility that both "func2" and "func3" are called during the same execution instance of the application. This could be the case in a typical network stack, where "tcp_send" and "udp_send" probably have the same function signatures, but can never be called from the same call site. Hence an incremental development process which combines inputs from the automated techniques and the programmer can improve the understanding of the application.

Further known techniques, such as abstract execution, flow analysis, *etc.* as well as techniques that will be developed in the future, can be used to gather more information and make the graph more complete. Our analysis can remain the same and take advantage of a graph which has more information – this will reduce the time for the analysis and provide more accurate results.

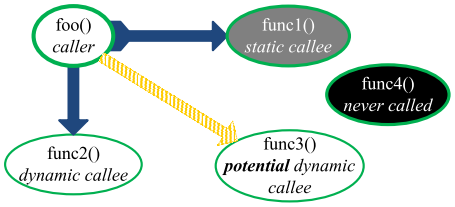


Figure 5. Timing graph created by application of various information gathering techniques

4.2. Graph Creation

The graph is actually created in stages. The green nodes in the graph which represent straight line execution within the same thread could be basic blocks in the CFG or even single functions (at a higher level). Traditional static timing analysis is performed to obtain the worst-case execution time (WCET) for this block of code. Edges obtained from static analysis as well as those obtained from the dynamic traces form the blue arrows, which are added next. Yellow edges are gleaned from the type information and are then added to the graph. Red edges are a result of dynamic tracing where calls to concurrency constructs are also traced. Further type analysis reveals other possible conditions that could be called/waited upon, based on the type signature of the previous callee, leading to black edges being inserted into the graph.

Before we start the analysis, the number of "possible" edges in the function call graph is large – any function can call any other function or signal/wait on any condition variable. With static analysis, some of the yellow edges are turned into blue edges, while some of the black edges are turned into reds. Also, since static analysis fixes one outgoing edge per call site, it also eliminates other yellow/red edges from the same call site, thus reducing the universe of possibilities. Further analysis (runtime tracing) turns some more of the yellows into blues and blacks into reds. However, this step doesn't reduce any edges, as there is no guarantee that each call site has only one outgoing edge – in fact, as we have seen in the case of function pointers, each call site could call many potential callees. Type information analysis, on the other hand, can prune some edges – it restricts the possible outgoing edges based on the type signature of the blue and red edges that have been discovered during the runtime analysis. Hence, some yellow and black edges are removed from the graph. Finally, domain knowledge can prune the graph further by removing impossible yellow and black edges from the graph. This multi-colored, pruned graph is used for our analysis that follows in the next section.

5. Timing Graph Transformations

In this section we shall enumerate certain fixed graph transformations which will help reduce the complexity of these timing graphs. The transformations will also help the programmer find interesting topological and programmatic properties. As explained in the introduction, embedded systems have severe resource constraints and executing parallel code could lead to a high number of context switches and a large number of threads, which ultimately lead to a high demand for stack space. Hence, reducing the program to obtain the *least* number of threads required for correct execution of the program aids in reducing stack pressure. We are also able to find the limits of serializability of the program. The transformation described in the remaining part of this section will help us achieve both goals. The larger goals for performing these transformations are:
(a) Find the *minimum* number of threads required for the application to function correctly.
(b) Aid in understanding the program behavior/structure and help system designers to optimize it. Specifically, this could find candidate spots for additional concurrency in the program, which makes the program more scalable. We could also find false parallelism in the program, where multiple threads must execute in a serial fashion for forward progress of the application.
(c) The ability to generate partitures, and from them the schedules of execution on a particular system, *automatically*

The timing graph and subsequent transformations could also be used, in the future, to achieve the following goals:
(i) Help in visualizing the program, so that system designers could gain an understanding of the true nature of the program.
(ii) Aid in increasing the parallelism of the program by finding spots that are synchronization bottlenecks – *i.e.*, many edges in the same spot.
(iii) Provide inputs for real-time schedulability analyzers and constraint solvers.
(iv) Aid in model-based testing of the application.

5.1. Assumptions

The assumptions (or rather the pre-conditions) for this work:

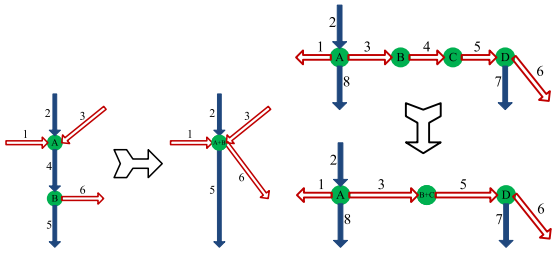
- A *strict partial order* is always maintained for the graph.
- The graph cannot have any deadlocks or race conditions (*i.e.*, no unguarded resources). This condition basically means that the program must be *correct*. Of course, benign races (such as atomic adds) are fine.

If the timing graph represents a real-time application, then certain additional constraints apply: *(a)* Loop bounds must either be statically known, or at least known prior to loop entry. *(b)* No code can be dynamically loaded - *i.e.*, all modules in the application must be statically known. We could relax this rule to state that we need to know all applications that could possibly be loaded – the system could then be treated as if everything had been preloaded (this would introduce some pessimism into the analysis). *(c)* The program must use a *finite* set of condition variables. *(d)* Either there must be no branches or forks around condition waits/signals, or all possible paths must be *correct* (deadlock and race free).

Note: our analyzer will not check for the correctness of programs, but will axiomatically assume the validity of programs and try to apply valid transformations.

5.2. Graph Pruning and Reduction

This section examines techniques used to reduce and simplify the graph. First, let us examine *graph pruning* techniques, which will help in reducing the "maybe" edges, either by transforming them to actual, known edges (blue or red) or getting rid of them entirely. Section 3 already introduced some methods of performing this pruning – for instance, dynamic traces prune some yellow edges to blues, *etc.* Other techniques used are:



(a) Assimilate nodes with only blue nodes into each other

(b) Merge consecutive nodes with only single red edges

Figure 6. Two Point-of-View Simplifications

(a) Black edges that result in "potential" deadlocks (with red or blue edges) are removed. The program is assumed valid unless proven otherwise.
(b) In the case of alternate yellow edges, pick the *worst* one of all. Hence, we pick the yellow edge with the largest WCET of all.
(c) If alternate paths exist and each one waits on different condition variables (or none), then the longest path must wait on a *union* of those condition variables – hence execution waits on all of the condition variables to be signaled, to proceed.
(d) If, in the face of alternate paths, the application signals different condition variables (or none) on each one of the alternate paths, then execution must wait for an *intersection* of those condition variables to be signal-led.

Remarks: Techniques *(c)* and *(d)* follow from a desire to preserve the worst-case behavior and strict partial ordering respectively. The former insures that all resources must be acquired before execution proceeds, while the latter guarantees that all branches are valid and does not result in deadlocks.

Graph reduction operations are broadly classified into two groups – *(I) point-of-view* simplifications and *(II)* simplifications that *restrict the partial order*. Of course, the transformations must not violate the invariants for the graph (section 3.2). *Note*: the examples in the figures indicate artificially created graphs to illustrate the transformation being performed; while these graphs are not extracted from actual code, it is entirely feasible that such situations could occur in real programs.

5.2.1. Point-of-View Simplification

The following two point-of-view transformations are illustrated in Figure 6: *(a)* Merge consecutive nodes connected by a blue edge, where either the blue node's source has no outgoing red edges or its destination has no incoming edges, or perhaps both. This is a simple concatenation of sequential code.

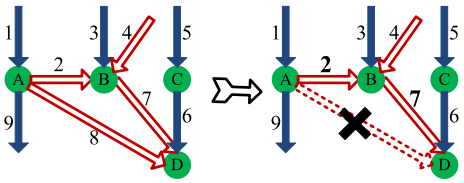


Figure 7. Remove direct red edges

(b) Two consecutive nodes which have single incoming and outgoing red edges can be fused into a single node. Nodes "B" and "C" were combined into node "B+C". This transformation corresponds to inlining the code from one thread into the other thread.
(c) Remove a direct red edge if a longer, indirect path consisting entirely of red edges exists between the source and destination nodes.

Remarks: Application of one or more of these simplifications does not result in a reduction of the nodes or edges in a graph – they are just dropped from the visualization. Hence while they may exist in the graph, for all practical purposes they may be ignored. Transformation *(a)* is shown in Figure 6(a), where node "A" has only one outgoing edge, which is blue, and node"B" has only one incoming edge ("4") which is also blue. Hence, they are merged into a single node ("A+B"). This transformation is equivalent to merging consecutive basic blocks (or function callee into caller) where no other dependencies exist for the caller or the callee. The WCETs of "A" and "B" can now be combined to form the WCET of the new block as follows:

$$WCET_{A+B} = WCET_A + WCET_B - pipeline\_interactions$$

where "pipeline_interactions" refer to the reduction in execution time due to the concatenation of the trailing edge of A and the leading edge of B [12]. Note: edge "4" is missing from the new graph, because it is actually included within the new node, "A+B".

The *pipeline_interactions* term in the above formula refers to the worst-case *execution* time for the flow of instructions through the pipeline. By performing the concatenation of nodes "A" and "B" the WCET of the combined node does not change – only the visualization and treatment in the graph becomes more convenient.

Figure 6(b) shows that the WCET of the combined node is the sum of the two nodes, with pipeline effects considered (as in transformation (a)). We are able to perform this transformation because the real dependency between "A" and "D" is not changed by the merging of the intermediate dependency (edge "4").

We can remove edge "8" between "A" and "D" (Figure 7) as an alternate path (A→B→D composed of edges "2" and "7") exists. This follows from the intrinsic transitive nature of a partial order. Of course, in a real-time system, we could interpret this as retaining the sequence of edges that exhibit worst-case behavior – two or more dependencies is worse than one, direct dependency.

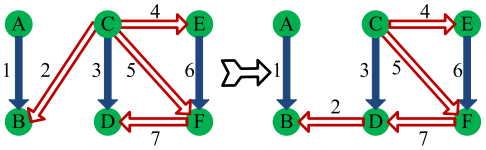


Figure 8. Move outgoing red edges to successor

5.2.2. Restricting the Partial Order

(i) Move all outgoing red edges of a node to its successor. A "successor" is defined as a node which is the destination of a blue outgoing edge from the current node.
(ii) Move all incoming red edges of a node to its predecessor. A "predecessor" is defined as a node which is the source of a blue incoming edge.

Remarks: These transformations aim to restrict the partial ordering for the graph. They actually change the edges in the graph – either by moving their source or their destination nodes, and sometimes both (though this will be done one at a time). These transformations are applied to every possible node, and their edges are transformed except of course in cases where they violate the graph invariants. Figure 8 shows transformation *(i)*, where nodes "B", "D" and "F" are successors to nodes "A". "C" and "E" respectively. From the figure we see that outgoing edges "2" (for node "C") is transformed to point *from* node D. Hence, the outgoing nodes for "C" are moved to C's successor "D". Note: we did not transform edge "5" because doing so would have created a deadlock with edge "7". Similarly, transforming edge "4" would have resulted in a deadlock as well ($4 \rightarrow 6 \rightarrow 7$).

In Figure 9, nodes "A", "C" and "E" are predecessors to nodes "B", "D" and "F" respectively. After transformation *(ii)*, we see that incoming edges "2" now point to node "C". We did not transform edge "5" because it would have created a deadlock with edge "4", thus violating an invariant. Similarly, transforming edge "7" would have resulted in a deadlock as well ($4 \rightarrow 6 \rightarrow 7$).

The above graph transformations are valid. They are analogous to known deadlock avoidance techniques. Transformation (i) is the same as releasing all locks held by the process at the same time, *i.e.*, at the end of the execution of the *outermost* critical section. Hence, we move all condition signals to the successor. The second transformation is the same as delaying execution of the critical section until *all* locks requested by the process have been acquired – *i.e.*, move all condition waits to the predecessor. These transformations can be performed recursively, thus ensuring that the critical section execution starts only after all locks have been acquired and will release all locks at the same time – at the end of execution. These two transformations could be further restricted with more invariants, in hard real-time systems, such as the deadline, startup time, period, phase, *etc.* *Note*: if we combine transformations (i) and (ii), we get the priority ceiling protocol (PCP) [10], assuming all resources had the same ceiling value.

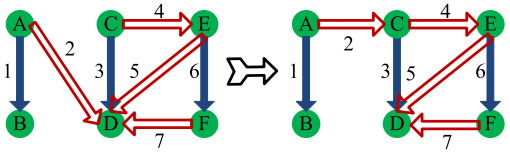


Figure 9. Move incoming red edges to predecessor

6. Outcome of Timing Graph Transformations

After iteratively applying the graph transformations (section 5), we obtain one of the following situations – *(a)* the graph is entirely serializable. In this case we can execute the entire program using a single thread. *(b)* The graph is non-serializable where, at the simplest level, it resembles the graph shown in 10(a). This is a graphical representation of a producer-consumer relationship, (Figure 10(b)), where "x" and "y" are condition variables. "W" signifies a condition wait while "S" represents a condition signal. Hence, Figure 10 represents the case where one thread (Thread 1, the producer) waits for another thread (Thread 2, the consumer) to send in a request which it then responds to. Note that the consumer is not able to make much progress, as it must wait for results from the producer to be sent back. Hence we can deduce that this simple program can make progress only if they execute on *two* separate threads.

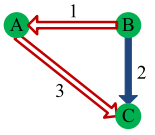
The graph shown in Figure 10(a) is the basic building block for larger, complicated, non-serializable graphs. Each of the green nodes could themselves be more complicated nodes which are constructed using the basic building block. Some examples are shown in Figure 11 (*Note*: the details of applying the graph reductions to obtain the above result are omitted here due to space considerations.) We are also able to calculate the minimum number of threads required for the application, from the reduced graph, using the following equation:

$$N_t = N_g - N_b \tag{1}$$

where N_t is the minimum number of threads; N_g is the number of green nodes and N_b is the number of blue edges in the graph. The reasoning for this equation is simple – each green node indicates a separate point of execution, and possibly a separate thread. Each blue arrow ties green nodes to one another, thus indicating that both must execute in the same thread. Once the effects of the blue edges have been thus accounted for, only interactions between threads (red arrows) remain and since we have performed all possible reductions on the graph, the existence of a red arrow indicates interactions across threads. The simple example shown in Figure 10(a) has three green nodes and one blue edge – hence, it requires two threads to execute. Similarly we can calculate the minimum number of threads for larger, more complex graphs (Figure 11).

6.1. Futures and Program Modifications

The final graph obtained after performing all reductions can be further reduced by simplifying modifications to the



(a) Smallest non-serializable graph

Thread 1	Thread 2
W(x)	S(x)
S(y)	W(y)

(b) Basic producer-consumer relationship

Figure 10. Outcome of graph transformations

program – *(a)* get rid of red edges (*i.e.* the interactions among threads. This is difficult to do because removing red edges results in modifying the inherent, expected behavior of the program and could easily make it incorrect. However, reducing red edges, when done with care, may make the program *more* parallelizable. *(b)* Convert blue edges to red edges. This is possible by using the *futures* [15] mechanism. If all the targets of blue edges (green nodes) are converted into futures, then consequently blue edges turn into red edges, and since futures are expected to execute at some point after they are invoked, the correctness of the program is maintained. From Figure 12 we see that when edge "2" is changed from blue to red, node "B" is converted into a future.

By turning nodes into futures, we *increase* the expressed concurrency of the program. If multiple processors are available then the futures can often execute in *parallel* (Figure 13(a)) to the extent allowed by the dependency graph (which is expressed as a *partiture*). It also increases the flexibility available to the scheduler – to decide when to execute the code in the future. Another important result is that because these nodes are now futures, they can execute independently of each other and only a loose order has to be maintained. Hence, they can even be executed *sequentially* on a single processor as long as the callees execute at some time in the future, *after* the callers. One such sequentialization is seen in Figure 13(b). The order of nodes "B" and "C" can be switched around to form another sequential schedule.

Thus, we see a surprising result – *by increasing the concurrency of the program, we can also increase its serializability*! This result may seem counter-intuitive, but has great potential. It indicates that by using the graph transformations outlined in the paper followed by the "futurization" of some nodes, the scheduler is given the flexibility to either parallelize the program for larger systems or sequentialize it for execution on small, constrained, embedded systems.

6.2. False Parallelism and Hot Spots

One of the goals of this work is provide a visualization of the reduced graph for the programmers to analyze, which could be achieved by feeding the graph structure from the "temporal timing analyzer" into the GLEE visualization tool [23]. This will help the programmer in weeding out *false parallelism* and problem spots in the program (*hot spots*). Hot spots are parts of the program where a large number of interactions could be concentrated (thus degrading the overall program performance). Hot spots are again identified by finding nodes which have an unusually large number of interactions centered around it – either incoming condition waits or outgoing condition signals, or even both.

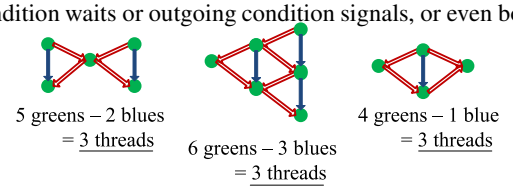


Figure 11. Multiple producer-consumers

What is an "unusually large number" is decided by the systems designers based on the actual application. Sometimes two could be large, and sometimes nodes can handle 20 interactions. While both of these problems are identified by visual inspection of the graph structure, it is not a difficult task to automate the process.

Each of the basic producer-consumer blocks (Figure 10) is an indicator of *false parallelism* in the program. While it requires multiple threads (at least two) for forward progress, the execution actually happens in sequential order – *i.e.*, B → A → C. No other order will work and none of these nodes can execute in parallel with one another. Note: Futures can *still* express concurrency without breaking existing relationships. Futures do not obliterate concurrency – in fact, they make the relationships explicit while moving the control logic out (into partitures).

7. Experimental Framework

The simulation environment used for the experiments and analysis is the Giano [7] simulator using the eMips CPU model, running the lightweight MMLite [14] operating system. Apart from the synthetic benchmark presented in Figure 4, the main benchmark used for analysis was the network stack from the MMLite operating system, compiled down to a single loadable module. The original MMLite network stack was an extension of the BSD implementation of the networking protocol. We developed a unique tool, the"*temporal timing analyzer*" (TTA), which aids in the creation of timing graphs. Other tools used in the information gathering process are – the MIPS compiler for Giano and the nm command line tool. The various steps used in the creation of the graph are – *(a)* a disassembly of the object code of the network stack is obtained using the MIPS compiler; *(b)* the list of functions in the network module is obtained using the nm tool; *(c)* both of the above are provided as inputs to the TTA, which, at first, creates a static control flow graph of the entire program. It is able to express information at the basic block level or even at a higher function level. The TTA is also able to determine the static dependencies among the various basic blocks/functions in the program – *i.e.*, it generates the green nodes and some of the blue nodes that go into the timing graph. It is also able to provide an estimate of the yellow and black edges in the graph; *(d)* a dynamic trace of the network stack running on Giano is obtained. The inputs are various webservice calls that trigger different functionality in the network. While this

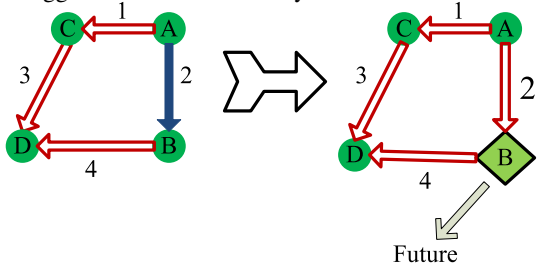


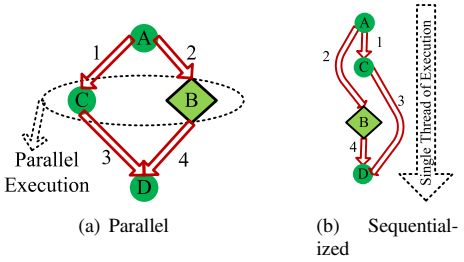
Figure 12. Converting Blue edges to Red – creating futures

does not guarantee complete dynamic coverage of the network stack, it is able to obtain quite a few dynamic dependencies (function pointers) and also match condition variables to their wait/signal sites. These traces, and the information gleaned from them, are fed into the TTA, where it is able to form a more complete picture of the timing graph. From the dynamic traces it is able to restrict some yellow edges to blue edges while completely doing away with some other yellow edges (as explained in section 4). From information on the wait/signals on condition variables, the TTA is also able to change some of the red edges to black edges and eliminate other impossible black edges. In fact, if we are able to determine the entire range of possible inputs for a function/application then tracing will be able to provide a complete picture of the dynamic behavior of the application; *(e)* type analysis is performed and the information is then fed into the TTA. This adds more information to the timing graph; *(f)* information from other sources, such as domain knowledge, abstract execution *etc.* (section 4) can also be plugged in to obtain a more comprehensive graph.

Any static timing analysis framework [5, 18, 19, 22] can be plugged in to the TTA to obtain the WCETs for the green nodes, after step *(b)*. We performed step (e) by hand and did not implement (f). The purpose of the experimental framework is to show that creation of the timing graph using information collected from various sources is entirely feasible, which it did, as we shall see in the results section (section 8). In fact, the design of the TTA enables us to provide it with information about the graph from any of the above mentioned (or even other sources), which will then be plugged into the graph to obtain a better understanding of the timing and runtime behavior of the application. The most interesting part about our analysis (graph reductions and subsequent observations) is that it can be performed on an incomplete graph as well as a graph which has all of its characteristics mapped. While the former will yield approximate results, the latter can yield precise results. These insights (on the state of the concurrency, sequentialization and resource constraints of the application) can greatly assist programmers and system designers.

8. Results

In section 8.1, we shall enumerate the results obtained by performing the transformations and analyses on the timing graph. Section 8.2 lists results obtained from the temporal timing analyzer which show that the process of creating the timing graph is a feasible one.



(a) Parallel (b) Sequentialized

Figure 13. Options for the Future

8.1. Graph Results

The following insights were obtained by performing the various analyses and transformations on the timing graph:
(a) The most important, and perhaps most surprising result, is that *increasing the concurrency* of the timing graph, using certain program modifications, resulted in *increased serializability*. This provides the scheduler with a lot of flexibility in creating the final schedule and tailoring it to the particular hardware system in use.
(b) Various graph transformations finally lead to three types of graphs – those that can be completely serialized, those that resulted in deadlocks or those that are constructed of basic producer-consumer relationships. From the last result, we are able to calculate the minimum resource requirements (threads and corresponding execution stacks) for correct forward progress in the program. This results in memory usage reduction in embedded systems.
(c) The analysis is able to direct the programmer's attention towards false parallelism and hot spots in the program.
(d) The final graph is the worst-case schedule possible for the program.
(e) From the graph reduction, we are able to minimize the number of context switches in the application.
(f) Inter-thread communication/dependencies are reduced.
(g) The transformations result in the smallest partiture.
(h) The graph helps programmers visualize and understand the application – this will tell them if their original design was correctly translated into code and perhaps, tell them if there were any deficiencies in the original design.

8.2. Temporal Timing Analyzer Results

Results from the temporal timing analyzer are tabulated in table 1. **Note**: These results are not intended to show the runtime performance of our analysis tool, but to show the effect of applying the combination of analysis techniques on the timing graph that we create.

The first column represents the type of information being analyzed, where "S" represents the static call information and "D" represents the dynamic call information. The second column lists the number of possible (yellow) edges. The third column represents the actual calls that were made (blue edges), while the fourth column lists the number of calls sites for each type of call - static or dynamic. The last column lists the remaining yellow edges after each type of analysis.

We first conducted experiments on a simple toy benchmark (Figure 4) to show that our ideas are feasible. This simple example shows that static analysis alone will not be able to capture the true nature of the program, as it will not

Call Type	Possible	Actual	Call Sites	Remaining
Toy (S)	5*5 = 25	1	1	0
Toy (D)	25	1	1	2
Network (S)	169,744	2386	412	0
Network (D)	169,744	76	76	31,312

Table 1. Graph edges based on static/dynamic information

be able to deal with calls through function pointers (func2). The benchmark has 5 functions and at the outset, it is possible that any function could call any other function (including itself) leading to $5 * 5 = 25$ edges. Once static analysis has been used, we see that func1 is called which leads to the creation of a blue edge. Dynamic analysis results tell us that another call (func2) was made. This adds a second blue edge, but there is a possibility that this dynamic call site could be used to call any other function as well. Type information tells us that only func3 has the same signature as func2 and has the potential to be called, while func4 has an entirely different type signature and will never be called. Assuming that none of the other functions made any calls, we reduced the number of edges from a possible 25 to 3 actual edges.

Another benchmark used was the network stack for MMLite (second half of table 1). This module contains 412 functions all of which, combined, are broken down into $4,886$ basic blocks. With such a large number of functions, the number of yellow edges, considering only static calls for all of these functions is $412^2 = 169,744$, as every function can potentially call every other function. Static analysis of the interactions among the functions tell us that only $2,386$ functions were called statically by all 412 functions. This converts $2,386$ of the yellow edges to blue edges and also eliminates all of the remaining $169,744-2,386 = 167,358$ ones.

Similar results are presented for dynamic calls. The remaining yellow edges were calculated as follows – each one of the 76 call sites can potentially call any one of the 412 functions leading to $76 * 412 = 31,312$ possible yellow edges. Type analysis now tells us that only functions which match the signature of these 76 callees can ever be called from these dynamic call sites, which will lead a further reduction in the number of possible edges. Domain knowledge can further reduce this figure, as some of these "potential" callees (with function signature matches) cannot be called during the same execution instance. Hence, the number of edges drops to a number which will be far less than $31,312$.

So, the initial estimates of $339,488$ potential edges (static + dynamic) have been reduced to a more manageable number which is in the tens of thousands, if not less – an order of a magnitude (or more) difference. Hence, we see that analysis of complex programs using our framework and graph transformation techniques is feasible – it is able to handle large programs, which to date, have been excluded from such analyses due to their inherent complexity. Note that we make no claims of reducing the *time complexity* of these programs. Such program were inherently considered to be un-analyzable due to the programmatic constructs they contain – function pointers for instance. In the past, analysis of such programs would not have been possible by any one of static analysis, dynamic analysis, type information, domain knowledge alone, but, in this work, was achieved as a result of combination of all these methods.

9. Related Work

Understanding the temporal behavior of programs is essential for system designers. This is more critical if the system being designed has other inherent constraints, such as the resource constraints seen in common embedded systems. Sometimes these systems have more severe constraints imposed on them (real-time systems) where advance knowledge of the behavior of the program becomes critical. We believe that the work presented in this paper is unique in that it is able to transform large applications into a graph representation on which transformations are applied to gather the "meaning" of the program, with the aim of making distributed embedded systems more scalable, primarily by creating models of the program based on our ideas of *partitures* and *futures*.

Andersson [2] studied the temporal behavior of embedded programs and also used dynamic traces to add more information to his analysis. This work differs from ours in that he creates a model of the application and uses model checking and regression analysis coupled with dynamic traces to reason about the temporal characteristics of the program. It stops with analyzing the impact of the behavior of the temporal characteristics of the program and does not provide any further insights like we do in section 6. We also deal with concurrent programs in our analysis.

The level of parallelism required by an application has been explored by Motus *et. al.* [20, 21]. They focus on model driven development, where an engineer writes a timing model (Q model), including educated guesses for minimum and maximum times of processes in periodic applications such as are found in telephone switches. The model is based on processes and channels. While our work facilitates writing the model by hand, it may also be used to analyze existing programs and does not require the processes to be periodic.

Henzinger *et. al.* [16] introduced the idea of compiler-driven feasibility checking of scheduling code. In their approach the compiler creates an executable, which represents the schedule, which is then attached to the end of the task. This schedule is executed and validated each time the task is invoked on specialized embedded hardware. Our work creates partitures and futures based on analysis of timing graphs. Also, compared to them, we are able to determine the levels of parallelism and potential problem spots in the application, which could prove beneficial to a large range of systems.

Previous work in real-time systems deals with either static or dynamic analysis. Recent, independent work [18] discusses a hybrid approach to performing timing analysis. Our work is similar in that it is also a hybrid approach which uses information for a variety of sources. The difference is that while our work is focused on finding the properties of interactions among threads in concurrent programs, theirs is focused on obtaining worst-case execution time results for modern, out-of-order processors. In fact, the results from that framework (the WCETs) can be plugged in to our temporal timing analyzer to obtain more precise results for ap-

plications running on modern processors.

10. Conclusion

In this paper, we used a combination of analysis tools and methods to glean information from programs and then combined the result into a graph. The graph represents a model of the temporal behavior of a program. This paper defines the graph coloring, invariants, and a set of valid transformations which are used to extract information out of the graph. One insight gained was that increasing the concurrency of the application can also lead to increasing the serializability. The graph could be output as a partiture that is usable as a manifesto of the program behavior as well as in scalable and distributed scheduling. In this paper we also extended the reach of static timing analysis to applications that were, due to their complexity, not previously analyzable for determination of worst-case behavior. This was done by combining static analysis with dynamic analysis based on traces and with other techniques such as type inference. The practicality of the graph generation and analysis methods were demonstrated with an implementation that was used to create a graph of an entire TCP/IP stack. The topological properties that the graph transformations reveal are useful in understanding and optimizing an application for variable levels of parallelism. The methodology presented here forms a solid base for further work in schedulability analysis.

References

[1] G. Agrawal, A. Acharya, and J. Saltz. An interprocedural framework for placement of asynchronous I/O operations. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 358–365, Philadelphia, PA, 1996. ACM Press.

[2] J. Andersson. Modeling the temporal behavior of complex embedded systems - a reverse engineering approach. Licentiate thesis, June 2005.

[3] G. Cugola and C. Ghezzi. CJava: Introducing concurrent objects in java. In *Object Oriented Information Systems*, pages 504–514, 1997.

[4] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[5] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, 2002.

[6] A. Forin. *Futures, in Advanced Language Implementation, Peter Lee editor*. MIT Press, Cambridge MA, 1990.

[7] A. Forin, B. Neekzad, and N. L. Lynch. Giano: The two-headed system simulator. Technical report vol. msr-tr-2006-130, Microsoft Research, One Microsoft Way, Redmond, WA 98052, 2006.

[8] D. Friedman and D. Wise. CONS should not evaluate its parameters. In *Michaelson and Milner (editors), Automata, Languages, and Programming*, pages 257–284. Edinburgh University Press, 1976.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The NesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, pages 1–11, June 9–11 2003.

[10] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. In *Proceedings of the 2nd International Workshop on Real-Time Ada Issues*, May 1988.

[11] R. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[12] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[13] J. Helander. Deeply embedded XML communication: Towards an interoperable and seamless world. In *5th ACM Conference on Embedded Software'*, September 2005.

[14] J. Helander and A. Forin. MMLite: A highly componentized system architecture. In *Proceedings of the ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, 1998.

[15] J. Helander, R. Serg, M. Veanes, and P. Roy. Adapting futures: Scalability for real-world computing. In *Real-Time Systems Symposium*, December 2007.

[16] T. Henzinger, C. Kirsch, and S. Matic. Schedule carrying code. In *Third International Conference on Embedded Software (EMSOFT)*, January 2003.

[17] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. Model-based software testing and analysis with c#. In *Cmbridge Univeristy Press*, 2007.

[18] S. Mohan and F. Mueller. Hybrid timing analysis of modern processor pipelines via hardware/software interactions. In *IEEE Real-Time Embedded Technology and Applications Symposium*, page (accepted), 2008.

[19] S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*, pages 233–242, Dec. 2005.

[20] L. Motus, R. Kinksaar, T. Naks, and M. Pall. Enhancing object modelling technique with timing analysis capabilities. *iceccs*, 00:298, 1995.

[21] L. Motus and M. G. Rodd. *Timing Analysis of Real-Time Software: A Practical Approach to the Specification and Design of Real-Time*. Elsevier Science Inc., New York, NY, USA, 1994.

[22] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[23] L. Nachmanson, G. Robertson, and B. Lee. Drawing graphs with GLEE. In *15th International Symposium on Graph Drawing*, September 2007.

[24] J. Preden and J. Helander. Auto-adapation driven by observed context histories. In *International Workshop on Exploiting Context Histories in Smart Environments*, September 2006.

[25] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.