Rapid Early-Phase Virtual Integration * †

Sibin Mohan[1], Min-Young Nam[1], Rodolfo Pellizoni [1], Lui Sha[1], Richard Bradford[2] and Shana Fliginger [2]
[1] Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana IL 61801
[2] Rockwell Collins, Cedar Rapids IA 52498
[1] [sibin,mnam,rpelliz2,lrs]@illinois.edu, [2] [rmbradfo,sefligin]@rockwellcollins.com

Abstract

In complex hard real-time systems with tight constraints on system resources, small changes in one component of a system can cause a cascade of adverse effects on other parts of the system. We address the inherent complexity of making architectural decisions by raising the level of abstraction at which the analysis is performed. Our analysis approach gives the system architect a rigorous method for quickly determining which system architectures should be pursued, and it allows the architect to track and manage the cascading effects of subsystem/component changes in a comprehensive, quantitative manner. The end product is a virtual architecture analysis *that systematically incorporates the inherent coupling among interacting system components that share limited system resources.*

1. Introduction

System architects face a fundamental dilemma when preparing competitive bids for complex system development projects, especially those with hard real-time constraints. The inherent complexity of these projects, coupled with the time pressures associated with the bidding process, force an architect to make subjective judgments involving significant uncertainties about the values of various parameters and their ultimate effects on the cost and performance of the system. On the one hand, being too "optimistic" in these judgments raises the risk of unpleasant surprises later in the development process. On the other hand, being overly conservative in accounting for these uncertainties can result in a losing proposal.

Figure 1 shows a high-level view of a generic system architecture decision process for real-time systems. The system architect must define an architecture that has sufficient levels of system resources, while avoiding unnecessary costs that increase the risk of lost business. As the development process proceeds, and more information becomes available, the system design must evolve to address any changes in assumptions. If the system cannot fit within the specified resource budgets, the system architect must add additional resources, which results in lost profits for the project. Problems and/or changes that occur during the analysis phase (right half of the diagram) could potentially result in significant changes at all levels of system design (e.g. high-level design, platform architecture, etc.), as shown by the dotted arrows.

The process of architecting such a system is highly complex due to the *non-linear nature of cross-effects introduced between various components such as the processor, caches, bus traffic and also application logic*. The entire system shows behavior akin to an ecological system where the *optimization of one parameter can have untold harmful effects on other parts of, or even the overall, system*[1]. In the absence of an *end-to-end tool chain and virtual integration techniques*[2], introduced in this paper, such long-range side effects will only be discovered during the system integration phase that will result in huge expenses. This is the reason why system integration costs for safety-critical systems are between 50–75% of the total system development cost [11].

Hence, we propose a set of analysis techniques to support decision making at the system architecture level. We address the inherent complexity of the architectural decisions by *raising the level of abstraction at which the analysis is performed*. Our techniques give the system architect the capability to select among various processor types, cache configurations, and bus topologies. With these techniques, which we instantiate in a *system integration toolset*, the architect can derive estimates of software application performance and communication network latencies and can incorporate these estimates into a performance analysis of the integrated system. The end product is a virtual architecture analysis that systematically incorporates the inherent coupling among the system components that inter-

* This work is supported in part by a grant from Rockwell Collins, by a grant from Lockheed Martin, by NSF CNS 06-49885 SGER, NSF CCR 03-25716 and by ONR N00014-05-0739. Opinions, findings, conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

† **Note**: All numeric values used are representative of generic avionics applications. However, these numbers have no connection to any existing, or future, Rockwell Collins or Lockheed Martin products.

1 Detailed examples follow in Sections 5, 6.2 and 6.3

2 In this context, "virtual" means that the analysis does not require a concrete implementation of hardware or software components.

1052-8725/09 $26.00 © 2009 IEEE
DOI 10.1109/RTSS.2009.48

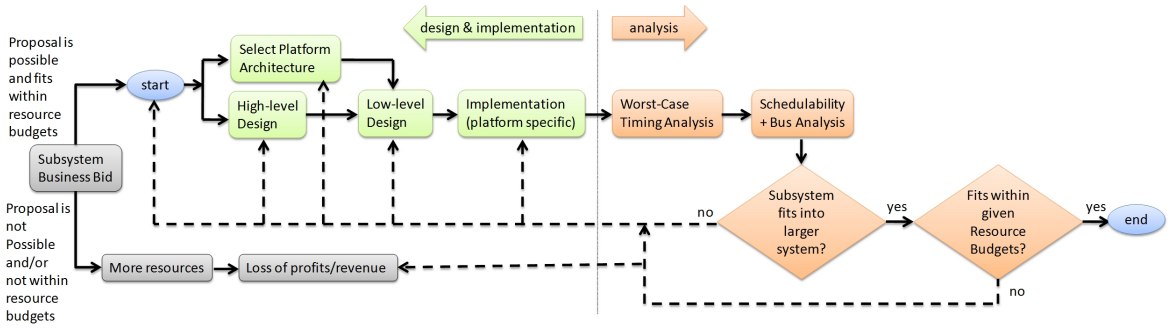

Figure 1: Overview of Current Design and Implementation Flow for Avionics Platforms

act while sharing limited system resources. The process of designing such a tool-set and integrating the various analyses, especially for complex domains such as avionics, automobiles, *etc.*, is extremely challenging, since it requires *expertise and an in-depth understanding of the complex interactions among differing domains such as scheduling, timing analysis, computer (processor) architecture, bus topology and flow design, caching, etc.*, not to mention the *application logic*.

In addition, these same analysis techniques have significant benefits during the system integration phase of the development process. In hard real-time systems with tight constraints on system resources, small changes in one component of a system can cause a cascade of adverse effects on other components of the system. These adverse effects can manifest themselves in two dimensions. First, a small change in the assumed performance of one component can lead to negative impacts on the performance of many other components, as could happen, for example, when one component requires more execution time than its specification assumes. Other components that receive inputs from this component are then delayed in receiving their inputs, hence their execution takes longer than expected to complete. These delays can ripple throughout a system, as each component is delayed and takes longer than expected to execute. Details for such an example follow in Sections 5, 6.2 and 6.3. Addressing these effects exposes the other dimension of the problem, namely, the cascade of adverse effects can lead to a significant increase in the amount of time, effort and costs required to reach a feasible system design.

In summary, this paper proposes a set of techniques to provide a system architect with the following capabilities:

C1 to analyze architectural trade-offs *prior to any concrete implementation* of the system (in sharp contrast to traditional analysis techniques) by using a combination of modeling languages and tools such as Simulink® and AADL;

C2 to perform *quantitative comparisons and rapid exploration* of the design space, including different choices for microprocessors and their options, different bus architectures and transaction types, and different software task and partition configurations; and

C3 to track and manage the *cascading effects* of subsystem/component changes during system integration in a comprehensive, quantitative manner.

To the best of our knowledge, this is the first work that aims to provide such support for system architecture decisions so *early* in the design phase.

In the next section, we describe a scenario in which a system architect wishes to assess the feasibility of incorporating an additional application into an existing system architecture. We have chosen a specific example involving a stall warning function for an avionics system. We use this example to demonstrate the limitations of existing approaches to timing analysis. In Sections 3 and 4 we describe our approaches for system architecture modeling and end-to-end analysis for virtual integration. Sections 5 and 6 describe our experimental framework and the results we obtain for our stall warning example. Section 7 discusses related work, and Section 8 concludes the paper.

$$v_{si} = \sqrt{\frac{2\left[W\left(\sin\alpha_w a_{xB} - \cos\alpha_w a_{zB}\right) - L_T - T\sin\left(\alpha_w + \xi_0\right)\right]}{C_{L_{v_{crit}}} \rho_o S_w}}$$

$where,$
$v_{si} : Stall\ Speed$
$W : weight\ of\ aircraft$
$\alpha_w : angle\ of\ attack$
$a_{xB} : accelerometer\ value\ in\ X axis$
$a_{yB} : accelerometer\ value\ in\ Y axis$
$L_T : lift\ produced\ by\ horizontal\ tail$
$\quad perpendicular\ to\ velocity\ vector$
$T : powerplant\ thrust\ in\ plane\ of\ symmetry$
$\xi_0 : thrust\ inclination\ angle$
$C_{L_{v_{crit}}} : critical(stalling)\ effective\ lift\ coefficient$
$\rho_0 : air\ density\ at\ sea\ level$
$S_w : reference\ wing\ area$

Figure 2: Stall Speed Calculations

2. System Composition Example: Stall Warning System

To study the effects of integrating a new subsystem into an existing architecture, we present an example of a *Stall Warning System Application* (SWSA) [8, 37]. The SWSA is a device on an aircraft used to *(a) detect* and *(b)* suggest/take corrective action to prevent an impending stall . An *aircraft stall* is a condition in aviation and aerodynamics where the *angle of attack* [8] increases beyond a certain point such that the lift starts to decrease. The angle at which an aircraft goes into a stall is known as the "critical angle of attack" and is dependent on the wing profile, its aspect ratio, and other factors. When an aircraft enters the stall mode, the pilot may notice that flight controls have become less responsive and may also notice some buffeting. This could cause serious problems and if not detected in time, lead to the aircraft crashing, with potential loss of human life. The fatal effect of an aircraft stall was in the news recently (while the aircraft involved in the incident did have a SWSA, it did not include an *audible* alarm, but this was not necessarily the reason for the crash) [16]. Clearly, the addition of an SWSA to an existing avionics platform can be quite desirable.

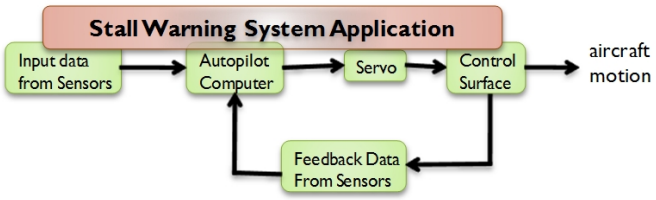

Figure 3: Overview of Autopilot subsystems affected by addition of Stall Warning System Application (SWSA)

There is a correlation between the "angle of attack" and the airspeed. This is convenient, as it allows us to calculate a *stall speed* below which an aircraft experiences a stall. "Stall speed" is the speed below which the aircraft cannot create enough lift to sustain the weight of the craft in $1g$ flight. Hence, the stall speed can be used as an indication of an impending stall, and the pilot can be notified and corrective action proposed/taken. Figure 2 shows the inputs and the formula for calculating the stall speed [37]. Once the SWSA finds that the aircraft has attained stall speed, it can send warning alarms to the pilot and/or take corrective action, *e.g.* by use of a "stick shaker" [5].

An implementation of the SWSA would require close interactions with various systems that form the avionics suite of an aircraft. Figure 3 shows how an implementation of the SWSA would require interactions with high-level components that form a part of an Autopilot application that includes input sensors, the main processor, servo motors, the control surface and perhaps even feedback data from various sensors. Hence, when a customer requests that an SWSA be added on to existing avionics applications, system architects must be able to make prompt assessments of whether it is feasible to do so. They may have some information about the resources available for such an implementation – *e.g.* from analyzing a similar application on another platform, they could reach the conclusion that 5% of the processor budget is available for implementing an SWSA. **Note:** In this paper, we concentrate on the *processing requirements* followed by related schedulability and bus delay analysis. Memory (RAM/Flash/Non-Volatile Memory/*etc.*) are left for future work. Sections 2.1 and 2.2 show how analysis (for the processor) would be conducted and limitations imposed on such analyses.

2.1. Analysis

Function/ Construct	WCET (cycles)
main	7549
stallSpeed	7219
sqrt_approx	855
pow_approx	141
cosine	1184
cosine loop	957

Table 1: WCET results for Simple implementation of Stall Speed calculations

Traditionally, analysis of the performance of a software application was carried out by use of timing analysis [36] methods to determine worst-case bounds. This requires the use of a processor model and an *actual implementation* of the application tasks. This is then timed by use of either analysis of execution traces (dynamic timing analysis) [7, 33] or a compile-time analysis of the control flow graphs of the code (static timing analysis) [22,36]. For our experiments, we implemented the stall speed calculations (Figure 2) in `C` and then used our static timing analysis framework [12, 23, 34], introduced in later sections (Section 4), to obtain worst-case execution times. Table 1 lists the WCET results obtained from analysis of this simple implementation. The first column lists the functions and loop constructs in the program and the second column lists the worst-case execution time (WCET) in cycles for each. The cycles for the `main` function represent the total WCET for the application and includes the WCETs for the other functions shown in the table. The `stallSpeed` function (called from within main) performs most of the calculations listed in Figure 2 and functions `sqrt_approx`, `pow_approx` and `cosine` perform the calculations for square root, power (a^b) and $cosine$, respectively (these functions are called multiple times from inside function stallSpeed). The `cosine loop` row shows the WCET for the main loop for the cosine function that is based on the Taylor (Maclaurin) series expansion. The processor model was a generic MIPS PISA [22] in-order architecture with a $64k$ 4-way set associative instruction cache and no data cache.

We then were able to show how this application integrates with an existing platform by use of schedulability and bus delay analyses (details in Sections 5 and 6). Hence, we are able to show how our combination of analysis techniques provide an understanding of the *end-to-end* performance during system integration.

2.2. Limitations

The biggest hurdle in performing end-to-end analysis for complex systems such as avionics is the need for a *concrete implementation*, as illustrated above. The system architect is unable to gauge whether the application fits into given resource budgets, since this analysis happens *late* in the design and development stage as shown in Figure 1. Hence architects are forced to make best-guess estimates, either from experience, or from studying requirements for other "similar" systems as explained before. If the architects get their estimates wrong, the result can be significant delays and loss of revenue, since now either *(a)* the project cannot be completed or *(b)* will require extra resources that were not budgeted. Additional complication arise since such new systems are typically not stand-alone in nature. They exchange data and interact with other important components (*e.g.* autopilot system in Figure 3).

Hence it is extremely useful to have an analysis suite that can provide information, *ahead of time*, on whether certain new applications can meet their budgets. This will provide system architects with the ability to make educated decisions even before an actual hardware platform or the detailed design of the system is available. Hence, they require *virtual integration* techniques, *i.e.* studying the effects of integration without there being an actual implementation, hardware or software. This is achieved, in part, by the use of a combination of our analysis techniques and existing *modeling* platforms such as Simulink [19] and AADL [10, 31] as explained in the following sections.

3. High Level Modeling

As mentioned in the previous sections, an estimate of the feasibility of integrating new functionality into existing real-time systems must often be carried out even *before* an actual implementation exists. There is a need for providing accurate estimates on whether new (sub)systems will fit into provided resource budgets early in the process. On the other hand, the right side (analysis) of Figure 1 requires detailed knowledge of both the software implementation and the hardware platform. To obtain tight WCET results during timing analysis, it is necessary to have accurate processor models as well as the application source code and/or the binary.

To overcome the contradictory requirements between early and accurate analysis, we propose the use of a modeling tool, such as Simulink [19], that also has automated code generation capabilities (Real-Time Workshop®, Real-Time Embedded Workshop®) [17, 18]. The steps for the analysis, then, are:

1. Formulate a *high-level, functional* model of the new subsystem with Simulink or other modeling tool.
2. *Automatically generate* code. In our case, C code using either Real-Time Workshop or Real-Time Embedded Workshop[3]. The difference between the two is that code generated by the latter tends to be "leaner" and more targeted towards embedded systems with processing and memory constraints. Either one can be used to generate code that is specific to certain processor platforms, such as Atmel, Freescale, *etc.*
3. Pass the generated code through the analysis suite, depicted on the right-hand side of Figure 1 to calculate estimates for worst-case timing, schedulability, bus delays, *etc.*

While the generated code may not be as efficient as code that can be developed specifically for certain application areas (avionics, automobiles, *etc.*), it is definitely valuable since, *(a)* the complete analysis suite can now be used to calculate the worst-case resource requirements for the application and to verify whether it is schedulable within given resource budgets, *(b)* it provides a mechanism to bound the worst-case effects of the application since this generated code will typically perform *worse* than code that is specially developed for the application, and *(c)* if it is noticed that there exists a somewhat systematic difference in performance between code that is automatically generated and code that is not, then we can account for this difference in our analysis.

Figure 4 shows the Simulink model we created that represents the Stall Warning System Application (SWSA) explained in Section 2. Input parameters are shown in green ovals on the left and on the top right. These values, such as the weight of the aircraft, accelerometer values, *etc.* (Figure 2) are provided by other subsystems. The "angle of attack" input parameter is shown in the red oval at the top left. The rectangular boxes represent either *subsystems* that perform specific calculations (large blue rectangles) or mathematical operations (white boxes, not filled), except for the five small orange boxes on top that represent mathematical constants (π, e, *etc.*). Figure 5 shows the details of one of the subsystems (TSinat) of the stall system model; it performs intermediate calculations based on the angle of attack and some other parameters.

We used Real-Time Embedded Workshop to create code for a "generic" processor. Our analysis techniques are still valid for specific processor types, as long as information that affects that particular type of analysis is provided as

3 For production systems, the resulting C code should be checked for compliance with avionics software guidelines.

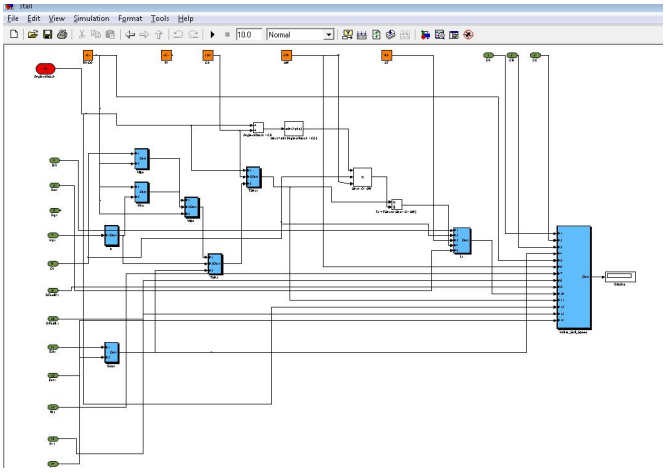

Figure 4: Simulink model for SWSA

part of the processor description *e.g.*, a processor model (pipeline structure, caches, *etc.*) must be provided for performing accurate timing analysis for that microarchitecture. The generated code (a few hundred lines) contains 18 functions: one for each of the 9 subsystems (Figure 4), 3 that manage the behavior of the stall subsystem (initialize, start, terminate), 5 functions that perform specific mathematical operations (32-bit multiplication, linear interpolation, *etc.*) and `main`. The number of functions, their types, their location, their memory management techniques (dynamic vs static), *etc.* all configurable through the Simulink code generation mechanism. Further details on the generated code, the constituent functions, *etc.* are presented in later sections.

This process of creating high-level functional models that can be used to automatically generate code helps fulfill part of requirement C_1 (Section 1). Capability C_1 is completely achieved by use of our remaining analysis components, as explained in Section 4.

4. End-to-End Analysis for Virtual Integration

As mentioned in previous sections, there is a need for performing *early* analysis of complex systems that includes multiple, complex parameters in order for system architects to come up with informed estimates of system performance. This need could result either from changes to, or from the addition of new (sub)systems, to existing platforms. In general, the problem is: given a set of (sub)systems, specified using a high level modeling tool as described in Section 3, and an architectural description of the available resource

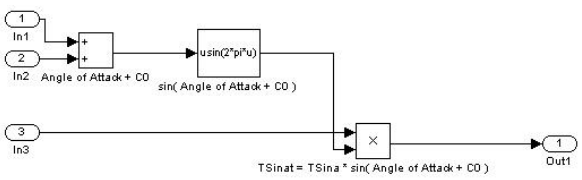

Figure 5: Detailed view of a Stall Subsystem (TSinat)

budget, can we feasibly schedule all (sub)systems on the allocated hardware architecture?

Answering this question is not trivial, because the end-to-end delay of a (sub)system depends not only on its computation, but also on the amount and type of communication flows exchanged in the physical processing node. This is made even more complex by the fact that embedded design is rapidly adapting COTS hardware components, such as those found in common Personal Computers (PCs), in an attempt to increase performance and reduce costs and "time to market". While COTS components can perform much better than dedicated hardware elements (for example, the PCI Express [1] peripheral interconnection found in common PCs is three orders of magnitude faster than the Safebus [13] employed used in the Boeing 777), they add significant complexity to the analysis. In particular, peripherals that need to exchange high throughput traffic such as video flows usually have DMA capabilities: they can autonomously initiate data transfers without direct CPU intervention. This means that the feasibility of adding the new (sub)system depends not only on CPU schedulability, but also on the schedulability of communication flows generated by peripherals.

To address this problem we describe, in this section, a novel framework (Figure 6) that derives end-to-end application characteristics that seamlessly integrates various analysis techniques. *(A) Timing analysis* is used to determine the WCET of each application (in isolation) based on a processor model. *(B) Schedulability analysis* is used to determine the feasibility of multiple applications executing concurrently on a CPU that uses partition scheduling [30] and also accounts for the effect of communication flows due to cache misses suffered by executed tasks. Finally, *(C) bus delay analysis* studies the mutual interference among communication flows and derives communication delays based on the model of the bus architecture used in the system. End-to-end delay for each (sub)system can then be obtained based on the computation delay computed in (B) and the communication delay from (C).

The ability to support *virtual integration* of hardware and software components is preserved since all analysis techniques are based on high level models of such components that are suited for early system-level specification: *(i)* code generated from a high-level Simulink model of the application (Section 3), *(ii)* the processor model(s) defined in software (both, for the static timing analyzer as well as for the schedulability analysis tool, the latter written in AADL) [10] and *(iii)* the bus architecture is also described in our analysis tool using AADL. By using an extended AADL model as the input for our tool, named *ASIIST* [4], annotated AADL designs of software tasks, commu-

4 Application-Specific I/O Integration Support Tool.

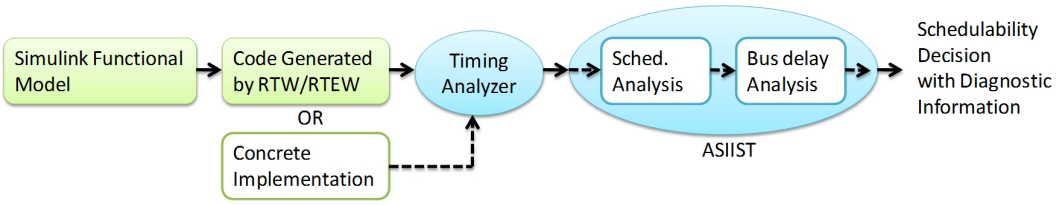

Figure 6: Overview of the End-to-End Analysis Framework

nication flows and hardware designs will be used to automatically generate schedulability equations. AADL provides the means for specifying the hardware and the software architecture for embedded systems. AADL provides textual and graphical interfaces for the users while giving APIs for tool developers and is being increasingly adopted for use in safety-critical domains such as avionics and medical devices.

Timing analysis, to determine the WCET of the application, is performed by use of our static timing analysis framework (Section 4.1). Schedulability analysis (Section 4.2) and bus delay analysis (Section 4.3) is done using a preliminary tool called ASIIST (Application Specific I/O Integration Support Tool) [24]. ASIIST reads in AADL models and can perform schedulability analysis with I/O cache fetch interference and I/O bus delay analysis for an IMA system configuration. A high-level view of our end-to-end analysis framework is depicted in Figure 6. In the figure, "RTW/RTEW" refer to Real-Time Workshop and Real-Time Embedded Workshop, respectively.

Note: If our analysis framework confirms that the application is actually schedulable, then it is *very likely* that a final implementation is also schedulable. Each step of our analysis (from the code generation in Simulink, through the timing analysis, the schedulability analysis and finally the bus delay analysis), is conservative in nature. Part of this "conservatism" is intrinsic in the high level modeling of the architecture: the behavior of many COTS components, such as peripheral buses, is regulated by a large number of complex parameters that are not reasonable for specification at the system design stage. An actual implementation would have code/hardware that is optimized, both automatically and manually, for all analysis of stages thus making it perform better than the conservative models we use and therefore, definitely schedulable. If the model is correct and its implementation is indeed a less optimized version of the final code then the analysis is guaranteed to produce safe bounds. So we say "very likely" (and not "guaranteed") due to the fact that we have no control on whether the model is correct and/or whether whoever implements the final code does it correctly (for instance, the final code performs worse than the automatically generated code).

This analysis framework aids in performing a rapid exploration of the design space (capability C_2 from Section 1). It, along with modeling techniques presented in Section 3, completes the capabilities required for analyzing architectural trade-offs *before* a concrete implementation is available (C_1). As shown in Sections 5 and 6.2, it also provides the ability to track and manage the cascading effects of subsystem/component changes during system integration (C_3).

4.1. Timing Analysis

Figure 7 shows the overview of our static timing analysis framework [12, 23, 34] used to obtain *safe* WCET values for applications, such as the SWSA in this case. Source files are fed as inputs to a compiler that extracts control flow and constraint information. A static cache simulator also extracts instruction hit/miss data for the application using information about the cache configuration. The control flow and constraints information, the caching categorizations as well as the machine dependent information (stages of the pipeline, instruction set, *etc.*) are provided as inputs to a timing analyzer (TA), that then derives WCET bounds. WCET values calculated by the TA are *safe* since they are pessimistic in nature and include overestimations of the execution time. Note that the TA calculates WCET values without information about program inputs.

The TA framework includes *three* different processor models (MIPS, SPARC and Atmel) and has the ability to analyze multiple instruction cache configurations (size, associativity, *etc.*). For our experiments, we used the MIPS generic processor model (based on the PISA instruction set architecture) and an instruction cache that is $64k$ in size and 4-way set associative. Our timing analysis framework did not consider a data cache (all data references miss and must go to memory) but existing [27] or new data cache analysis techniques can be easily integrated to improve tightness bounds for our results.

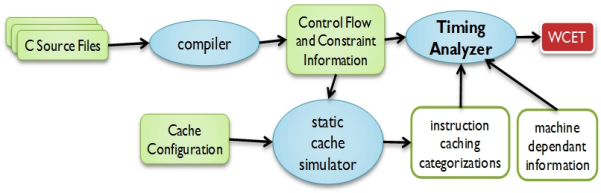

Figure 7: The Static Timing Analysis framework

The source files fed as input to the compiler in the first stage, can be an actual implementation, as in the case shown

in Section 2.1 or those generated from Simulink models (Section 3). This situation is depicted in Figure 6.

The Simulink models and hence, the generated code and related timing models, are as detailed as the designer/architect wishes them to be. If a high-level (and quick) analysis is required then the model (and everything that follows from it) can be less detailed. If, on the other hand, the designer(s) wish to spend time and effort to obtain a detailed model of the system, then that can be accommodated as well.

4.2. Schedulability Analysis

Schedulability analysis is carried out using the information obtained from the TA. Once the WCET of each application is measured in isolation so that no external bus traffic exists, it is specified in the AADL model that is considered to be an input to ASIIST. When multiple applications run together on a cached CPU, interference from communication flows must be considered: since main memory is a shared resource, when a CPU cache controller tries to fetch a cache line after a cache miss it can be delayed by a peripheral reading/writing in memory. In turn, this will delay the application, increasing its actual WCET. The WCET increment can be very significant, up to 44% in our experiments [25].

Our analysis methodology is able to compute safe upper bounds on WCET increases given information about both, the task under analysis as well as the peripheral traffic in main memory. The task is statically divided into a series of S sequential *superblocks* $\{s_1, \ldots, s_{S_i}\}$. Each superblock can include branches and loops but must be executed in sequence. We assume that the WCET (without peripheral traffic) and worst case number of cache misses in each superblock is known; in Section 6.2 we show how this information is obtained. As for peripherals, we assume that a bound on the total amount of traffic in main memory is known as this can be computed according to the bus analysis detailed in Section 4.3. Given this information, the algorithm introduced in previous work [25] is able to compute the maximum cache delay in each superblock and therefore derive a new modified WCET for the task. Using this modified WCET ASIIST applies the schedulability algorithm [30] to determine the schedulability of the applications sharing a CPU in an IMA system. ASIIST will show a graphical table of the components and their schedulability and parameters of interest. Users can change certain simple parameters that do not require architectural changes to test new results without reloading the whole model. Figure 8 is an example of the output that ASIIST shows.

4.3. Bus Delay Analysis

As mentioned in the previous section, bus delay analysis is also performed by using ASIIST. A logical data flow that is specified as a connection between two processes in AADL will, in reality, will go through multiple hardware components to get to a destination. *E.g.*, the PCI interconnection [1] has a tree structure where peripherals transmit on bus segments connected by bridge components. If the source and destination applications run on different CPUs located in separate systems, the logical flow could even go through a network.

Depending on the type of the hardware used, data traffic is regulated according to different arbitration types and COTS protocols. A designer can also provide various I/O configurations (Figure 9) to improve system performance. System performance is dependent on hardware flows that are derived from the logical data flows and I/O configurations. To specify such hardware flows new properties should be defined in AADL so that it is capable of representing the various possibilities that could occur for the same logical data flow. Still, the model must remain simple enough so that users can quickly make changes. Figure 10 shows an example of such a specification where the hardware flow is described as a sequence of hardware components which the data actually passes. A property is defined for each bus protocol specific transaction that is used. This enables and restricts ASIIST to analyze data flows that are supported by the tool and can be guaranteed for hard real-time properties such as delay bounds.

In particular, we previously showed [24] how ASIIST can be used to derive delays for data flows on the PCI bus. Our employed analysis methodology is based on the theory of network calculus [6] that is flexible enough to model a wide variety of arbitration models and buffering schemes. Initial delay bounds can be obtained using general assumptions although they can be somewhat pessimistic. By refining the architectural specification during the design process the user can obtain progressively tighter bounds. Hence, we see that system architects have the ability to quantitatively assess alternate hardware designs and quickly verify their feasibility and schedulability (C_1) at an early stage (C_2).

5. Experimental Framework

As shown in Figure 6, we start our analysis from one of two sources:

- The Simulink functional model of the Stall Warning System Application (SWSA) that is then used to generate C source code using Real-Time Embedded Work-

Executing Components

Discription	ID	Name	CPU Percent Runtime	Utilization	Util Marg...	Schedulability	Exec time(sec)	Period(sec)
Processor		proc0				Schedulable		
Partition		vm0	1	0.0023699...	0.0026425...	Schedulable		
process		PR_AP_R		0.0023699...	0.0026425...			
thread		Pitch_Rate_Thread		0.0023699...	0.0026425...		0.00002369999...	0.01
Partition		vm1	3	0.01185	0.0032636...	Schedulable		
process		PR_FG_R1		0.01185	0.0032636...			
thread		Pitch_Cmd_Thread		0.01185	0.0032636...		0.00118499999...	0.1
Partition		vm2	2	0.0014363...	0.0086140...	Schedulable		
process		STALL_WARNING		0.0014363...	0.0086140...			
thread		Stall_Speed_Thread		0.0014363...	0.0086140...		0.00001436300...	0.01

Figure 8: Adding a New System (SWSA) to an Existing Application

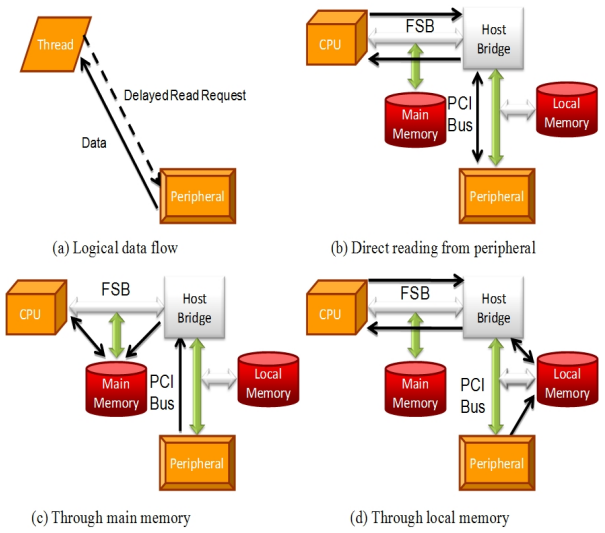

Figure 9: I/O Configuration Examples in AADL

shop. We used RTEW instead of RTW since it generated "leaner" code with fewer overall number of lines.

- An actual implementation of the application. For this purpose, we used the simple implementation described in Section 2.1.

The source files are then fed to the Timing Analyzer that generates the WCET for the application. This information is fed to the ASIIST analysis framework that uses partition scheduling [30] for the CPU. For our experiments, we use the following scenarios that may occur when adding a new application, such as the SWSA, to an existing IMA system [30]:

I. The new application can be assigned to a *new* partition when the CPU has enough slack in execution time. Such a situation would occur for fault tolerance and safety. The addition of a new application should not result in the failure of existing partitions. Side-effects from the failure of the new application can be limited to its own partition. For this situation the sizes of other partitions must be altered to make space for the new partition. This requires a re-analysis of the schedulability of *all* partitions.

II. The new application can be added to an *existing* CPU partition when it has enough execution time slack. The size of this partition must be adjusted and will affect other tasks that reside in it. Sharing a partition has the benefit of using intra-partition communication methods (buffers, blackboards, events) [2] that require less overhead than inter-partition communication (global messaging).

```
Properties
-- (b) Direct reading from peripheral
hardwareflow::Read_Actual_Connection_Binding => ( reference CPU,
    reference FSB, reference Bridge, reference PCIBus,
    reference Peripheral ) applies to connection in modes (option1);
hardwareflow::PCI_Read_Type => Delayed_Read applies to connection
    in modes (option1);

-- (c) Through main memory
hardwareflow::Read_Actual_Connection_Binding => ( reference CPU,
    reference FSB, reference SysMem ) applies to connection
    in modes (option2);
hardwareflow::PCI_Read_Type => Delayed_Read applies to connection
    in modes (option2);
hardwareflow::Write_Actual_Connection_Binding =>
    ( reference Peripheral, reference PCIBus, reference Bridge
    reference FSB, reference SysMem ) applies to connection
    in modes (option2);
hardwareflow::PCI_Write_Type => Posted_Write applies to connection
    in modes (option2);

-- (d) Through local memory
-- Similar to (c) but with option3 and to LocalMem
```

Figure 10: Hardware Flow Specification in AADL

Parameters	FGS	AP	SWSA
WCET	1185 μs	23.7 μs	14.363 μs
Init. Partition Size	1 %	3 %	N/A
Period	100 ms	10 ms	10 ms
Sum Input Data	N/A	N/A	24 Bytes
Sum Output Data	N/A	N/A	4 Byte

Table 2: Application Parameters

In cases (I) and (II) above we also show how *secondary effects*, such as cache interference, can cause the entire system to *fail* schedulability tests—even if all tasks were deemed to be schedulable while considering only execution time. These problems would typically have been discovered *late* in the implementation phase. Such analysis would be extremely difficult without the use of an analysis framework such as the one we present here. The difficulty would further increase if the analysis needs to be performed at an early stage. We also show how we can then modify system parameters—*quickly*—and test their feasibility, thus making the entire system with the new application schedulable again. We do not show the trivial case where there is insufficient slack for incorporating the new application.

The system model used for our experiments is a Flight Control System (FCS). Fight Guidance System (FGS) and the Autopilot (AP) are the two main applications of a FCS that runs on general computing units. FGS compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. When engaged, AP receives these guidance commands and calculates the actual pitch and roll rate commands to be sent to the control surfaces. We assume that they share the CPU with other applications and thus would have less than 100% of the CPU budget available. In Table 2 we enumerate the fixed parameters of the applications including the SWSA. The deadline is assumed to be the same as the period for all of these tasks.

6. Results

We now present the results from our end-to-end analysis framework. Section 6.1 presents the WCET values obtained for the code generated from the Simulink models. Sections 6.2 – 6.3 present the results of the analysis for the scenarios illustrated in Section 5.

6.1. Timing Analysis Results

Table 3 lists the WCET results for the stall speed calculation code generated from Simulink and Real-Time Embedded Workshop (RTEW). The first column lists the function names, the second lists the WCET estimates for each function (that includes time taken by callee functions).

Function Name	WCET
main	13040
stall_initialize	112
rt_OneStep	12572
stall_step	12448
stall_Eqtot	225
stall_H	2597
icsi_log	467
my_exp	812
pow_approx	339
stall_Vifps	1553
sqrt_approx	661
stall_Rho	345
stall_Vtfps	490
stall_TSina	508
stall_TSinat	6476
ldexp_approx	209
pow_approx	42
my_floor	23
my_fmod	142
look1_iu16lu32n31ys16_even9lcf	2775
plook_u32u16u32n31_even9c_f	448
intrp1d_is16s32_u32u32n31_l_f	1175
mul_s32_s32_u32_sr31	700
mul_wide_su32	473
stall_Lt	295
stall_VSikts_Stall_Speed	1407
stall_terminate	12

Table 3: WCET Results for Stall Code generated from Simulink/RTEW

The WCET for *main* presents the total worst-case time for the entire task/program. Function *main* calls functions *stall_initialize*, *rt_OneStep* and *stall_terminate* that represent the initialization, one pass at calculating for stalling speed and the termination of the task respectively. In a real-time system the *rt_OneStep* function would repeat in a cyclic loop and each iteration of the loop would form one "instance" or "job" that executes on the system. The *rt_OneStep* function calls functions *stall_step*, *stall_Eqtot*, *stall_H*, *stall_Vifps*, *stall_Rho*, *stall_Vtfps*, *stall_TSina*, *stall_TSinat*, *stall_Lt* and *stall_VSikts_Stall_Speed* that represent various subsystems, defined in the Simulink model (Figure 4), that perform intermediate calculations [37]. The output of the *stall_VSikts_Stall_Speed* function is the stall speed for the given angle of attack (α_w from Figure 2). This can then be checked against the current airspeed to verify whether a stall condition is imminent. Functions *icsi_log*, *my_exp*, *pow_approx*, *ldexp_approx*, *my_floor* and *my_fmod* are approximations of mathematical functions, *viz.*, logarithm, exponent (e^x), power (a^b), ldexp ($x.e^x$), floor and fmod. Functions *look1_iu16lu32n31ys16_even9lcf*, *plook_u32u16u32n31_even9c_f* and *intrp1d_is16s32_u32u32n31_l_f* are used for linear interpolation and table lookup that implement an approximation of the mathematical $sine$ function. Similarly, functions *mul_s32_s32_u32_sr31* and *mul_wide_su32* perform multiplication for long numbers. All of the above approximations are included to obtain a close estimate of the WCET for the task that includes the time spent in the *C* math library.

The functions in Table 3 are listed in the order in which they are called by their respective parents. Functions invoked by *rtOneStep* are called one at a time and *once* per each job instance.

The WCET results for the stall speed calculation code that is generated using Simulink/RTEW performs worse than the simple implementation depicted in Section 2, since Simulink/RTEW is more pessimistic in generating code for a particular platform. The code, while of fairly good quality, can definitely be hand-tuned to be more optimal for the hardware platform where the application finally executes. Even though the WCET values are larger and the code more pessimistic, this information is still useful since it forms an approximate upper bound for the actual application, as explained in Section 4. The WCET information described here forms the input to the next stage of analysis, *viz.* schedulability and bus delays.

6.2. Scenario I: Integration Into New Partition

For our first experiment, let us examine the process of adding a new partition to incorporate the new SWSA application into the existing Flight Control System (FCS). Schedulability analysis shows that it is schedulable when the size of the new partition is at least 2% of the CPU. Table 4 (also Figure 8) shows a *part* of the outputs from ASIIST, post-schedulability analysis. The first column represents the partitions, the second column shows all the threads in each partition, the third shows the CPU budget (as a percentage of total CPU time), the fourth column represents the utilization required by that particular partition, the next column shows how much of the utilization is left over after schedulability analysis (for each partition) and the final col-

Partition/ Process	Thr.	CPU	Util.	Util. Margin	Sched.
vm0(AP)	PR	1%	0.00237	0.002643	Y
vm1(FGS)	PC	3%	0.01185	0.003264	Y
vm2(SWSA)	SS	2%	0.001436	0.008641	Y

Table 4: Adding SWSA to New Partition

umn indicates whether that partition is schedulable or not. The table shows three partitions, "vm0", "vm1" and "vm2" that contain the AP, FGS and SWSA subsystems respectively. "PR", "PC" and "SS" are the Pitch_Rate, Pitch_Cmd and Stall_Speed threads, in each of these partitions, that perform their respective operations/calculations. Table 4 shows that all partitions are schedulable, with a positive utilization margin, when considering execution time slack. ASIIST calculates the utilization bound [30] for performing the schedulability analysis.

The new application (SWSA) will add I/O traffic that could interfere with other existing applications, even though they exist in different partitions. ASIIST can then recalculate the WCET for each task that includes time spent waiting for data that is delayed due to additional interference from the newly added task. ASIIST implements the analysis presented in Section 4.2 and computes a modified WCET that includes cache delay effects. To apply the analysis, each application or task must be divided into a set of sequential superblocks with given WCET and maximum number of cache misses. As described in Section 6.1, during each instance of the SWSA task the *rt_OneStep* function calls a series of "sub"-functions in sequence. Therefore, we can simply create one superblock for each such sub-function. The superblock's WCET is set to be the total WCET of the function, and the number of cache misses is set to the number of memory references for the function. Since we do not have a data cache, we can perform the analysis by counting the number of data references to the memory subsystem (Table 5). This information is derived as part of the timing analysis phase.

Function (Superblocks)	Mem. Refs.
stall_Eqtot	10
stall_H	23
stall_Vifps	17
stall_Rho	19
stall_Vtfps	42
stall_TSina	17
stall_TSinat	63

Table 5: Memory References for Superblocks

ASIIST can now perform the complete analysis by considering secondary effects, *i.e.*, interference on the bus due to additional memory requests introduced as a result of adding the new SWSA partition. Table 6 shows the results of performing such analysis. The execution time increases for all three tasks, FG, AP and SWSA. Hence, we see that *FG (residing on a* **different partition** *from SWSA) is no longer schedulable*. The utilization margin for "vm1 (FGS)" shows a negative value, thus indicating that the threads in this partition have exceeded their execution time budgets. Hence, even though the partitions are schedulable based on the execution time/utilization characteristics (Table 4), when one considers secondary effects such as bus and cache interference, the system could become unschedulable. This is a counter-intuitive effect that is hard to gauge without the assistance of such an analysis framework such as the one we present here. These problems would have been discovered late in the implementation/testing phase, thus leading to slipped schedules and and monetary loss. Hence, we show how capability C_3 (from Section 1) is provided by our analysis framework. On increasing the partition size of FG (to "4" in Table 7) all partitions become schedulable again.

Partition/ Process	Thr.	CPU	Util.	Util. Margin	Sched.
vm0(AP)	PR	1%	0.002903	0.002109	Y
vm1(FGS)	PC	3%	0.017877	**-0.00276**	**N**
vm2(SWSA)	SS	2%	0.004363	0.005723	Y

Table 6: Unschedulable after applying Secondary Effects

Partition/ Process	Thr.	CPU	Util.	Util. Margin	Sched.
vm0(AP)	PR	1%	0.002903	0.002109	Y
vm1(FGS)	PC	**4%**	**0.017877**	**0.002326**	**Y**
vm2(SWSA)	SS	2%	0.004363	0.005723	Y

Table 7: Modifying partition size for Schedulability

6.3. Scenario II: Integration Into Existing Partition

We now try to add SWSA to an existing partition. Let us assume that we implement it as a new thread in partition where the FGS subsystem resides. Table 8 shows all three stages of the analysis similar to that depicted in Section 6.2. The system, starts off being schedulable while only considering execution (Table 7(a)), becomes unschedulable on considering secondary interference effects (Table 7(b)) and is schedulable again after the partition size was modified (Table 7(c)). The partition size of FG had to be increased to 5% to make it schedulable. Note that there is always a minimum granularity that is used for partition sizes. We used 1% for this minimum granularity so that if the major cycle length is 100 milliseconds a partition size of 1% would be 1 millisecond.

7. Related Work

Our end-to-end analysis methodology is similar to the concept of *Platform-Based Design* (PBD) [28]. A platform is a library of (usually parametric) components. A platform instance is a set of library components selected to generate a concrete design. In a PBD design flow, the designer first specifies the system functionality using an implementation-independent description language. The designer then selects

a suitable platform and performs *mapping* of functional elements to platform components, thus creating a platform instance. Similarly, other model-driven design methodologies have been several of which [3, 14] support formal verification of modeled application behavior. Our analysis framework is different since we can actually perform the analysis without having an actual hardware platform (or even a software implementation). We can also target secondary effects of integrating new applications into complex, existing ones in hard real-time domain such as avionics. While recent work [26, 29, 32] shows the use of model-based engineering in related areas, our work is different in that we are able to provide analyses for systems that have hard real-time constraints and we have been able to bring together tools that. perform timing analysis, schedulability analysis and network/bus delay analysis. This integration helps fill a void in the real-time domain. We are also able to use it to perform analysis ahead of time so that avionics vendors are able to estimate requirements early in the design phase.

Methods to obtain upper bounds on execution time range from dynamic observation [7,33] to static analysis [22,36]. Past work mainly focuses on static analysis techniques, since dynamic techniques have been shown to be unsafe [33]. Recently, hybrid methods [4,20,21,35] that exploit the advantages of both static and dynamic analysis have been proposed Most of these techniques require an actual implementation of the application and do not consider the analysis of code generated from high-level functional models. De

(a) Schedulable

Partition/ Process	Thr.	CPU	Util.	Util. Margin	Sched.
vm0(AP)	PR	1%	0.00237	0.002643	Y
vm1(FGS)		3%	0.013286	0.001827	Y
	PC		0.01185		
	SS		0.001436		

(b) Unschedulable after applying Secondary Effects

Partition/ Process	Thr.	CPU	Util.	Util. Margin	Sched.
vm0(AP)	PR	1%	0.00237	0.002643	Y
vm1(FGS)		3%	0.022203	**-0.00709**	**N**
	PC		0.017877		
	SS		**0.004326**		

(c) Schedulable Again after Modifying Partition Size

Partition/ Process	Thr.	CPU	Util.	Util. Margin	Sched.
vm0(AP)	PR	1%	0.002903	0.002109	Y
vm1(FGS)		**5%**	**0.022203**	**0.003114**	**Y**
	PC		0.017877		
	SS		0.004326		

Table 8: Adding SWSA as New Task to Existing Partition

Oliviera *et al.* [9] perform timing analysis on code generated by Matlab/Simulink. They used a simple application to control electric motors, implemented on a DSP. Their processor architecture was much simpler compared to ours and they had significant difficulties in trying to capture WCET values for individual functions. Both of these issues do no apply to us, as indicated by the framework and results in Sections 5 and 6. Kirner [15] also performed timing analysis on code generated from Matlab/Simulink. They used a combination of "flow facts" and the implicit path enumeration technique (IPET) to perform their analysis. While these techniques are able to capture the WCETs of applications, they do not combine with existing schedulability and bus analysis tools to perform complete analysis as we present in this paper. Note: Research in timing analysis is orthogonal to our current work, since any existing/new TA tools can be combined with our analysis framework to obtain results specific to different processor families.

8. Conclusion

In this paper we presented a set of analysis techniques to support decision making at the system architecture level. Using these techniques, the architect can rapidly derive estimates of the performance of software applications and hardware components and incorporate these estimates into an integrated analysis for the whole system. The end product is a virtual architecture analysis that systematically incorporates the inherent coupling among the software applications that interact while sharing the limited system resources. Our analysis approach gives the architect a rigorous method for quickly comparing possible system architectures. Hence we have delivered capabilities $\mathbf{C_1}$ and $\mathbf{C_2}$ mentioned in the introduction.

These same analysis techniques have significant benefits during system integration. In hard real-time systems with tight constraints on system resources, small changes in one component of a system can cause a cascade of adverse effects on other components of the system. Our analysis techniques enable the system architect to track and manage the cascading effects of subsystem/component changes in a comprehensive, quantitative manner, thereby delivering capability $\mathbf{C_3}$. To the best of our knowledge, this is the first work that aims to provide such support for system architecture decisions so early in the design phase.

References

[1] PCI-SIG. http://www.pcisig.com/specifications/.

[2] ARINC. Avionics application software standard interface, ARINC specification 653, 1997.

[3] F. Balarin and et al. Metropolis: An integrated electronic system design environment. *IEEE Computers*, 36(4):45–52, 2003.

[4] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

[5] N. T. S. Board. Systems group chairpersons exhibit pertaining to airspeed, attitude, and stall protection system (sps). http://www.ntsb.gov/Events/2005/Pinnacle/exhibits/322636.pdf, 2005.

[6] J.-Y. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[7] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. 1997.

[8] L. J. Clancy. *Aerodynamics*. John Wiley and Sons, 1975.

[9] R. S. De Oliveira, M. V. Linhares, and R. B. Borges. Timing analysis of automatically generated code by matlab/simulink. *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 4575–4580, November 2006.

[10] P. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems. In *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*, pages 1206–1211, Oct. 2006.

[11] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 2002.

[12] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, Dec. 1995.

[13] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace Electronic Systems Magazine*, 8:34–39, March 1993.

[14] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[15] R. Kirner. Extending optimising compilation to support worst-case execution time analysis. *Dissertation, Technische Universität Wien*, 2003.

[16] J. Lowy. Low-speed warning system might have helped pilots. The Buffalo News http://www.buffalonews.com/260/story/670415.html, 2009.

[17] Mathworks. Real-time workshop. http://www.mathworks.com/products/rtw/.

[18] Mathworks. Real-time workshop embedded coder. http://www.mathworks.com/products/rtwembedded/.

[19] Mathworks. Simulink – simulation and model-based design. http://www.mathworks.com/products/simulink/.

[20] 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*, pages 285–294, 2008.

[21] S. Mohan and F. Mueller. Merging state and preserving timing anomalies in pipelines of high-end processors. In *IEEE Real-Time Systems Symposium*, Dec. 2008.

[22] 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.

[23] S. Mohan, F. Mueller, D. Whalley, and C. Healy. Timing analysis for sensor network nodes of the atmega processor family. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 405–414, Mar. 2005.

[24] M.-Y. Nam, R. Pellizzoni, L. Sha, and R. M. Bradford. ASI-IST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009*, June 2009.

[25] R. Pellizzoni, B. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. *Real-Time Systems Symposium, 2008*, pages 221–231, 30 2008-Dec. 3 2008.

[26] J. Porter, G. Karsai, P. Völgyesi, H. Nine, P. Humke, G. Hemingway, R. Thibodeaux, and J. Sztipanovits. Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation. *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, pages 20–34, 2009.

[27] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 71–80, Apr. 2006.

[28] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[29] E. Senn, J. Laurent, and J. Diguet. Multi-level power consumption modelling in the aadl design flow for dsp, gpp, and fpga. In *International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB 2008, in conjunction with MODELS 2008)*, 2008.

[30] L. Sha. Real-time virtual machines for avionics software porting and development. In *RTCSA*, pages 123–135, 2003.

[31] Software Engineering Institute. SEI open source AADL tool environment (OSATE). http://www.aadl.info/aadlinfosite/OpenSourceAADLToolEnvironment.html.

[32] O. Sokolsky, I. Lee, and D. Clarke. Process-algebraic interpretation of aadl models. In *Ada-Europe*, pages 222–236, 2009.

[33] 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.

[34] R. White. *Bounding Worst-Case Data Cache Performance*. PhD thesis, Dept. of Computer Science, Florida State University, Apr. 1997.

[35] J. Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, University of York, May 2008.

[36] R. Wilhelm and et al. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, Apr. 2008.

[37] D. S. Wilner and G. A. Bever. An automated stall-speed warning system. Nasa technical memorandum, NASA, 1984.