Embedded System Validation for Polymorphous Computing Architectures
WHITE PAPER

Ted Bapty
Institute for Software Integrated Systems, Vanderbilt University

Background

Performance verification is a key capability in the successful design, construction, and fielding of real-time embedded systems. Current system design technology has very limited capability for verifying systems from designs and simulations. While formal verification and simulation should be employed as possible, validation (and especially continuous, 'in-vivo' validation) is critical to fielding of reliable systems.

Verification of systems typically involves a structured testing process. The system is instrumented and exposed to simulated external events. The instrumentation records the system's response, and these responses are compared to design parameters and system specifications.

The instrumentation can take several forms:

1. Hardware instrumentation is typically the least performance-intrusive method. Examples of hardware instrumentation are:
   - Logic Analyzers or Emulators: These monitor a processor and its busses to gain a detailed view of the processor's execution. Instruction sequences and detailed traces can be initiated from user-programmed triggers. A highly accurate time-base permits the tester to know the exact internal details of the system under test. The drawback of this approach is expense. It requires hardware modification, or at least connection of complex test fixtures into the system. Often, the test fixtures are larger than the system, and their size and complexity prohibit use in the actual operation of the system. (Note that this is changing somewhat, as processor vendors incorporate emulation hardware into their chips.) Another drawback of this method is its labor-intensive nature. A significant amount of labor and expertise must be devoted to determine the proper instrumentation points. Likewise, a larger effort is required to analyze the (potentially huge volumes of) data. The collection of data and its analysis are an iterative process, typically performed only during the design process.
   - Network/Communication Monitors & Bus Analyzers: These devices can monitor the flow of data between distinct processors across a network. Since the device 'snoops' on the network (i.e. it plays no part in the transactions), it has no impact on the timing/performance of the system under test. Typically, these devices are used to gather general statistics, such as bus loading, collision rates (for a shared-media bus), however selected packets can be logged for later analysis. These devices are, like emulators, relatively expensive, laboratory-grade instruments, and are not often fielded in embedded systems.
2. Operating System Instrumentation consists of statistics collected during the normal execution of a system. Operating Systems can keep count of the number of times processes execute, the run-times for each process, and the average loading/idle time of the processor. Overall, the OS instrumentation provides a cursory level of system performance, which cannot be used to validate particular system constraints. The measurement process impacts performance of the application, but typically can be lumped into the OS overhead.
3. User-specific instrumentation involves the developer manually instrumenting code. The user, with knowledge of the application implementation and the system specifications, inserts application-specific code to measure and validate system performance. The measured parameters can include any information, such as timing and execution frequency, or internal parameter value ranges. The measurement routines can be configured to measure only the parameters that are necessary, minimizing the impact on application performance. Local processing of the measured data can be used to minimize the amount of storage required to accumulate information. Overall, this approach will provide the most relevant, application-specific information to the designer/verification process.

The major drawback is its labor-intensive implementation. If the code becomes part of the executing system, the diagnostics itself must be verified with the same care as the production code. It becomes a significant expense, which increases with the number of constraints to be checked. When system constraints evolve, the verification must be manually updated. If the verification code is not part of the final system, then the data from the measurements may not apply to the un-instrumented production code.

The above verification techniques have various advantages and disadvantages. While hardware approaches are very accurate and non-intrusive, they cannot be used in the target system, due to the added cost/size/weight and in the volume of data produced. Application-specific software instrumentation can efficiently gather information, however, it can be extremely costly to implement and maintain.

Future Validation of Embedded Systems

These techniques will require updating as system development tools improve. Much as using a compiler has changed the way software is coded and debugged, high-level tools will change the way performance is validated. Compilers abstract the basic machine architecture, so software developers no longer debug by looking at register contents and address ranges, but rather with source-level debuggers. There is no longer a human-readable mapping from hardware components to software. As systems are synthesized from high-level descriptions, the mapping from software textual code to the system description will be obscured. We will require that the validation design be specified at a high-level, and translated into instrumentation. Also, like a source-level debugger translates registers back to ‘C’ source variables, the validation facilities in a synthesized system will need to translate into high-level system descriptions.

Future design tools for embedded systems will need to support the automatic synthesis of validation code. The synthesized code must have the following attributes:

- Continuous Validation: The system validation must be done over the life of the system. While developmental testing should verify all nominal conditions, the system response to extraordinary events that only occur during operation of the system must also be monitored. Consequently, the validation will become part of the fielded system.
- Accuracy: the results must be accurate in verifying the target constraints. While software approaches will satisfy most of these requirements, hardware support will greatly assist.
- Derivation from System Constraints: While user-defined constraints must be supported, the system must be capable of constructing validation activities from high-level system specifications. Since specifications describe the important features of a system, it is exactly these attributes that must be monitored and verified in the synthesized validation.
- Efficiency: The system must make parsimonious use of measurements. Embedded systems are typically resource constrained. Addition of validation activities will increase the demand for resources, since these validation activities will be part of the final system. Minimizing the validation processes will minimize the resources required.
- Minimal Impact: The validation must not interfere with the satisfaction of system specifications. This also brings up the problem of validating the instrumentation used in validation. This will necessitate hardware support.

In this effort, we have studied the validation of systems assuming a model-based software synthesis approach to embedded systems. We have based the approach on tools developed at ISIS for the Adaptive Computing Systems program.[BAP1]

Model Based Software Synthesis of Embedded Systems

The Adaptive Computing Systems (ACS) environment divides the design process into four major categories:

1. **Behavioral Modeling:** In this first category, the operational adaptive behavior is defined. The designer can specify the operating modes of the system, the legal transitions between modes (and the conditions for transition), and the specifications for system operation while in each operating mode.
2. **Algorithm/Structural Modeling**: In this category, potential algorithms are described. The algorithms define signal flow specifications to compute required system outputs.
3. **Resource Modeling**: The resource models describe the hardware available for construction of the system. This consists of physical processors, devices, and the interconnection topology.
4. **Constraint Specification**: The above modeling categories are augmented and linked together with a constraint framework. The constraints allow user-defined interactions to be specified, establishing linkages between properties in one category and objects in the same or another category.

For the purposes of validation, the Algorithm/Structural Modeling is most significant, although Constraint and Resource Modeling are also necessary. Note that each behavioral mode can be validated independently, as these represent distinct, non-overlapping modes of operation.

Synthesis of Validation

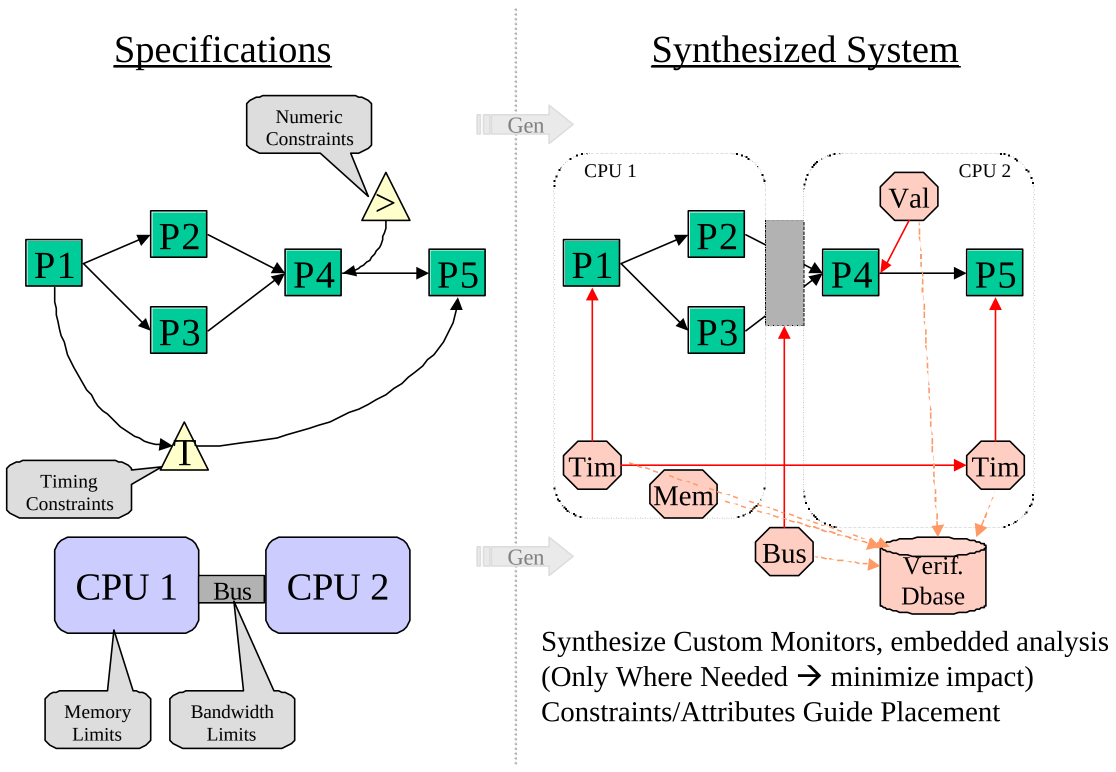
Capturing information in this form is very advantageous to the validation process. These benefits are as follows:

1. System performance requirements are explicitly expressed in the form of *Constraints*. (In a code-based design, these would be implicit in the programming. Automatic extraction would be very difficult.) The constraints tell us exactly what features we must verify.
2. Hardware capabilities are explicitly expressed in the form of *Attributes*. The hardware capabilities can be viewed as performance bounds during operation. For instance:
   - Memory usage must never exceed certain limits (i.e. free space must be available to run the next task).
   - Bus bandwidth must not exceed the physical media rates.
   - Processor loading must stay within prescribed limits.
3. The algorithmic structure, and thus the measurement points, are directly accessible via the models. Constraints are expressed in terms of points within the algorithm structure, making it relatively simple to decide where to insert instrumentation.

The generation process is summarized in the diagram below. Specifications, on the left, define the functionality of the system. Data flow defines the information processing functions and their data interactions. These data flows are annotated with constraints. Two types of constraints are shown here, as triangles. A timing constraint restricts the maximum latency of the system to respond to an input to P1 with an output at P5. A value range restriction on the output of P4 is also specified in the models. Synthesis of instrumentation to validate these constraints is relatively straightforward.

- Time constraint: A time-stamp module is inserted prior to P1. This module will read the system timer and annotate the data packet with this time, as the starting point. A data collection/statistics module is inserted on the output of P5. This will compute the delta time, and compare it to the constraint. The slack time will be computed and simple statistics will be accumulated (e.g. minimum, maximum, and average slack, number of timing violations, and average late time). With user guidance, additional domain-specific data can be collected about the late data packets.

- Level/value constraint: A simple data stream splitting component and a customized value-checking component will be added to the data flow. Statistics will be collected, along with data logging if needed and if resources are available.



Hardware verification will require extension of the operating system/kernel. Memory 'water-level' markers can be monitored and statistics computed on memory-critical processors. The facilities will be enabled based on the computed memory load from the model synthesis process, choosing processors that are within user-defined limits of the hardware capability.

Bus bandwidth issues are similar. For critical links, we must be able to instrument the connection with overall statistics: average utilization, duty cycle, max burst-time, max dead-time, max fifo length, and potentially others. The analysis of these statistics must be done as part of the runtime system, since the volume of data can be very large. In order to achieve this, the OS must support a plug-in for application specific monitoring. Many of the functions require access to internal data structures within the kernel. Clearly, the user must not have direct access to these structures, for performance and safety reasons. Instead, a programmable facility must be included into the OS, able to accept monitoring application-specific instructions. The instructions for this plug-in must be synthesized from the models. This is a research issue to derive these functions and the programming instructions.

Synthesis of hardware instrumentation is also highly desirable. Reconfigurable architectures offer the opportunity to embed customized instrumentation into a system. These concepts are being explored by BYU in the ACS program[BYU]. Logic probes and logic analyzer modules are being developed that could be inserted into a synthesized hardware design, with the proper input connections. Post-processing of this data is a more complicated problem. Specific triggering within the module is a first level of data reduction, effectively extracting only points of interest. Pathways from the modules must be synthesized to get data from the hardware. The typical approach is to generate a scan-path with a JTAG output. These are typically interfaced into an emulator in an interactive debugging environment. Since we are concerned with autonomous, minimal-additional-hardware operation, that approach will not suffice. Here, we must synthesize the an autonomous monitoring process, to gather statistics, compress the information, and log to a storage media.

Interpretation of Validation Data

The results of the validation data acquisition must be translated back to meaningful information to the designer. Since the design is represented in the high-level system models, the validation results must be mapped there as well. Since the validation acquisition instrumentation is generated, it is a straightforward process to reverse this mapping.

Within the models, the validation information can be used multiple ways:

1. Errors or constraint violations can be highlighted for the designer, focusing attention on problem areas.
2. Data can be used to configure simulations. The major problem in resolving a problem is reproducing the state of the system. If sufficient information is available, we can replicate the state in a system simulation, and generate the inputs to a discrete-event simulation package.
3. The models used for synthesis rely on accurate information for components (e.g. min/max execution time, data rates, etc). Feedback from the validation will help in producing better quality synthesis.

Conclusion

This white paper presents ideas for use in systems synthesis. Validation has been and will continue to be a major factor in the construction of embedded real-time systems. The validation problem will increase with the complexity of the target systems. System complexity is expected to skyrocket in the near future.

Implementation of these techniques are planned under the Polymorphous Computing Architecture program. We expect this effort to build upon these ideas with extensions for low-level architectural performance. The validation opportunities within a polymorphous architecture are vast, since resources can be dynamically associated to data processing or validation.

References

[BAP1] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S.: “Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems,” VLSI Design, 10, 3, pp. 281-306, 2000.

[BYU] Programming Approaches for Runtime Reconfiguable Systems, http://splish.ee.byu.edu/arpa/Summary.html