Uniform Execution Environment for Dynamic Reconfiguration

T. Bapty, J. Scott, S. Neema, and J. Sztipanovits
Vanderbilt University / Institute for Software Integrated Systems
Department of Electrical and Computer Engineering,
Box 1826 Station B, Nashville, TN 37235
bapty@vuse.vanderbilt.edu

Abstract

Modern high-performance embedded systems face many challenges. Systems must function in rapidly changing environments. Power/size constraints limit hardware size, while extreme performance requirements demand algorithm-specific architectures. Hardware architectures must structurally adapt to achieve high performance with changing algorithms. Reconfigurable computing devices offer the promise of architectures that change in response to the changing environment. The primary difficulty in this approach lies in system design. A model-integrated approach is used in the design capture and synthesis of these systems. The target systems are built on a heterogeneous computing platform including configurable hardware, ASIC and general-purpose processors and DSPs.

This project is a DARPA Adaptive Computing Systems funded effort, involving close cooperation with US ARMY/AMICOM.

1. Introduction

This research is motivated by the requirements for design and implementation of high-performance embedded applications with highly constrained implementation requirements. One such example is adaptive missile automatic target recognition (ATR) systems. Current ATR systems have extremely large computational requirements, on the order of 10 GOP/second in some modes of operation. Image sizes are currently small (128^2) with a moderate frame rate (30 frames/sec). Future image sensors will expand the available image data to 512^2 at 100's of frames/sec. Processing of this input data is vital to the guidance of the aircraft and thus must meet hard real-time requirements.

ATR systems must be physically small, less than 1 cubic foot. Weight is also a major consideration. These factors require that component utilization be maximized as much as possible for selected hardware. ATR systems also require special attention to power consumption.

The system behavior is highly mode-dependent. Different algorithms are required for different operating points. The performance requirements also change during the systems operational lifetime. During some processing modes, power is severely limited, due to heat dissipation and/or battery availability.

In order to achieve these requirements, the system architecture must be tuned to the algorithm. When multiple algorithms are required at different times, the designer has two choices: Choose a compromise set of algorithms and a single compromise architecture that is adequate to achieve performance; or choose an optimal set of algorithms and implement an adaptive architecture that can reconfigure to achieve performance on each algorithm. Reconfigurable computing offers the potential to achieve the second, more optimal approach.

In the design process of many complex systems it is not obvious which implementation choices will yield a good balance between power usage, hardware size, and system performance. Also, design cycles for these systems tend to be long. By the time a design is completed the hardware platform that the system was designed for may become obsolete. An adaptive system is needed to support a hardware platform that may be evolved to include current state-of-the-art technology without complete redesign.

In a single mission, an ATR system traverses a large number of operation modes (target acquisition, target tracking, aim point selection, etc.). Each mode can have very different processing requirements (latency/throughput/accuracy), resources, and operational constraints. Different modes may have a different computational structure. Reconfigurable computing offers a method for maximizing the utility of the computational platform by adapting architectures to the changing needs of the system. This provides reuse of components over time.

The research described in this paper implements a methodology and associated set of tools for the design and implementation of reconfigurable, high-performance, resource-constrained embedded systems [1]. The target architectures may be constructed of general-purpose CPU's with associated software and/or direct-implementation hardware processing elements. The CPU's include DSPs (Digital Signal Processors), conventional RISC/CISC processors, and optimized architectures that execute software processes. The hardware processing elements include fixed-function devices such as an ASIC FFT implementation and programmable logic devices such as FPGA's (Field Programmable Gate Arrays).

FPGA's have the ability to implement arbitrary hardware functions. FPGA's can also be reprogrammed quickly to completely change the function(s) implemented. Currently FPGA's are evolving quickly to have faster configuration times and larger capacity. These properties make FPGA's a

good choice for high-performance processing applications where flexibility is needed [2]. Multiple processing functions may be allocated to FPGA's and can change as the design evolves or the processing functions may be completely changed on-the-fly during system operation as processing requirements change.

The design of reconfigurable computing systems represents a significant challenge to the engineering process. System complexity skyrockets, since we are no longer designing a single system on a fixed architecture. We must now consider the design to be an integration of subsystems, each subsystem representing a phase of the system, with different subsystems existing over time. Additional complexities arise from the need for all subsystems to share the same physical implementation.

The Institute for Software Integrated Systems is developing an approach for managing the complexity in designing reconfigurable systems. Experience with Model-Integrated Computing (MIC) has shown itself to be successful in comparable situations [3]. The MIC approach involves the following steps:

- Use **Multi-aspect, Domain-specific Modeling Environments** to capture requirements, design methods, and resources in a format that is customized to the problem and its formalisms.
- Develop **System Synthesis tools** for converting the models into executable artifacts.
- Develop a **Runtime Execution Environment** for supporting the execution of the generated system.

2. Modeling Concepts

A multi-aspect, domain-specific modeling paradigm was developed to capture *all* information necessary to model and synthesize a system. This information includes system computation requirements, computation algorithm(s), and available system resources. The modeling paradigm has three aspects: a *structural aspect*, a *behavioral aspect*, and a *resource aspect*. Each of these aspects is defined in a graphical language customized for the domain of adaptive computing systems.

2.1 Structural Aspect

The structural modeling aspect is used to describe the processing algorithm structure. To manage system complexity, the concept of hierarchy is used to structure algorithm definition. This logical composition of systems using component subsystems has proven effective design structuring for large, complex systems.

Another significant technique allows the specification of multiple algorithm architecture alternatives for a given task. When alternatives are used, the algorithm structural models describe a huge number of potential design implementations. The large design space gives environment the freedom to search for and select an implementation that meets the specified requirements and fits within available resources.

The algorithm is modeled as a dataflow structure with the following objects: *compounds*, *primitives*, and *templates*. A primitive is a basic element representing the lowest level of processing that is modeled. A primitive maps to a processing object that is to be implemented as either a hardware function or a software function.

A compound is an aggregation object that may contain primitives, other compounds, and/or templates. These components can be connected within the compound to define the information dataflow. Compounds provide the hierarchy in the structural description that is necessary for managing the complexity of large designs.

A *template* object is used to capture the representation of a choice between multiple design architectures. These alternatives can be either compounds or primitives, allowing subhierarchies of designs. In a model, a template can be used to capture different *algorithm* alternatives or different *implementation* alternatives.

In signal processing, many types of tasks can be accomplished in multiple ways, for example in the spatial or the spectral domain. Both approaches will achieve the same basic results but with vastly different algorithm designs. Other algorithm characteristics can vary as well, such as latency and/or accuracy. In the spatial domain a filtering function can be achieved by performing a standard mathematical convolution. In the frequency domain, the function is achieved by performing a FFT, followed by a multiplication with the spectral representation of the filter, followed by an inverse FFT. In this case, the spectral method is more efficient as the filter order increases, resulting in a faster, smaller system. On the other hand, since the FFT is a block-based computation, the latency is at least a block-length.

Another use of templates is to model multiple implementation alternatives, i.e. different ways a processing function may be implemented. For example, a convolution can be computed in software running on a DSP, in software running on a network of multiple DSP's, in a hardware function in a FPGA, or in a dedicated hardware ASIC solution.

Algorithm alternatives allow the model of the system to capture design possibilities. Each of these alternative methods has different performance attributes and different hardware requirements. The selection of the best alternative depends not only on the hardware that is available, but also on whether the hardware is to be time-shared, and what hardware is already allocated to support the processing algorithms that are required for operations in different modes.

For the high-level designer, algorithm alternatives allow a virtual separation of algorithm from implementation. Typical algorithm design requires the engineer/physicist to consider the hardware details of the underlying architecture to achieve an efficient implementation. The ultimate effect is that the resulting algorithm reflects the hardware structure. This leads to the creation of highly non-portable, technology-specific designs. System upgrades to use more modern

technology require a bottom-to-top redesign. Algorithm alternatives promise to separate the algorithm from the architecture, to postpone the implementation decisions to a much later step in the design process. This approach should greatly simplify technology migration efforts.

2.2 Behavioral Aspect

The behavioral aspect defines the *modes* in which the system will operate and specifies the conditions under which mode changes occur. The modes and state transitions are specified in the behavioral aspect by a state transition graph.

The event expression that can trigger a mode change is defined by the *transition rules*. A transition rule is a Boolean equation composed of event variables. When this expression is satisfied the transition from one mode to another is enabled and system reconfiguration is to take place.

The behavioral modeling aspect is linked to the structural aspect by the means of *References*. Each mode references a model in the Structural Aspect that defines the processing algorithm that is to be operational in that mode. The references allow a single algorithm to be applied to any number of system states, or allow all states to have separate processing structures.

The behavioral modeling aspect also allows the specification of minimal timing requirements and maximal power usage. The power characteristics are specified using attributes of the models, in which the designer can enter a maximum power limit. Maximal system delays can be specified for any pair of input and output ports on the structural model.

2.3 Resource Aspect

The resource aspect defines the hardware platform available for the target. The top-level hardware system is a *Network* of components. Network components are either processor elements (DSPs, standard RISC/CISC processors), programmable logic components (FPGAs), or dedicated hardware ASIC components for fixed functions (such as FFT computation).

The components are constructed using *cores* and *ports.* Every processing element must contain one core. The core object captures the necessary performance attributes of the processing element such as clock speed, memory, and other resources. A core represents the processing element. A port represents a physical communication channel. Ports have associated protocols and specific pin assignments. Connections between processing elements are created by connections between ports.

3. Runtime Execution Environment

The runtime environment must support implementation platforms with the following attributes:

- Heterogeneity: Optimizing the architecture for performance, size, and power requires that the most appropriate implementation techniques be used. Implementations will require software (implemented on RISC and DSP processors), configurable hardware on FPGAs, and a mix of ASIC components.
- Low Overhead/High Performance: the runtime environment must minimize overhead, since overhead results in extra hardware requirements.
- Hard Real-Time: The target systems have significant real-time constraints.
- Reconfiguration: The execution environment must allow hardware and software resources to be reallocated dynamically. During reconfiguration, the application data must remain consistent and real-time constraints must be satisfied.

These issues must be addressed at multiple levels. At the lowest level, the hardware must be capable of reconfiguration. Software-programmable components have excellent inherent hardware support, since the main point with software is the ability to change system function based on memory contents. Internal hardware structures are designed to restrict dangerous conditions that could damage hardware. In FPGA's these safeguards do not exist and must be enforced manually or by the design/runtime infrastructure.

At a slightly higher level, the internal state of the software must be managed under changing tasking. Modern operating systems have evolved to support the flexible implementation of multiple tasks on a single processor in the form of time-sharing and/or multitasking. Typically, however, these configurations are not suitable for low-level efficiency and hard real-time conditions.

Moving to the next level, we must consider the operation of multiple functional units operating in a cooperative network. Finally the issues of application-specific requirements must be addressed, to allow the peculiar requirements of specific numerical performance and timing to be achieved in an implementation. The issues with consistency are addressed in the next section.

3.1 Hardware Consistency

The runtime system must avoid operational defects during a reconfiguration event. Hardware consistency can have many negative effects, from temporary loss of performance in an operational mode to hardware damage and total, permanent system malfunction. Typically, these deal with specific issues involving interfaces between hardware processes and/or devices. Some of these defects are illustrated in Figure 1 below.

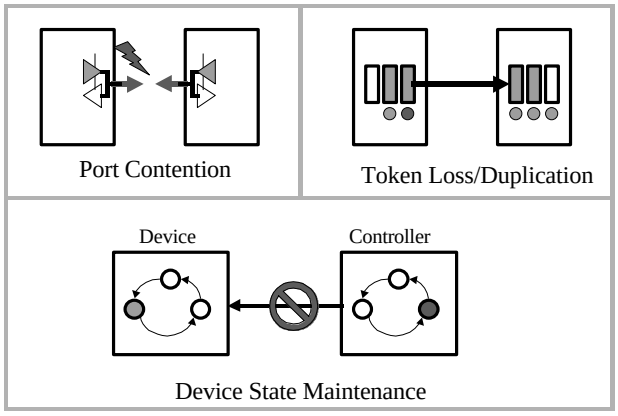

Figure 1: Hardware Consistency after Reconfiguration

Port contention occurs when bi-directional ports are improperly initialized or if an improper/inconsistent design is implemented. In this case, two connected drivers are enabled. If resistance is sufficiently low, permanent physical damage can occur to the circuits.

Token loss or duplication results from incorrect initialization or a loss of communication integrity. Tokens represent the status of empty or full slots in a communication interface. An extra token on the sender side can cause too much data to be sent, resulting in a FIFO overrun. A lost token can cause communication starvation, resulting in a system deadlock.

Device state maintenance refers to the control of a complex external hardware device, such as an attached processor or storage device. In controlling an external device, the controlling computational component must maintain an accurate representation of the device's state. If a reconfiguration occurs during a state transition within the device, or if the reconfiguration modifies the computational component's representation of the device, there can be a state mismatch. This can result in improper commands being sent to the device, or in a deadlock where both components are waiting on each other for triggering events.

These three examples show some of the potential hazards that can occur when the hardware device is improperly reconfigured. Runtime reconfiguration support must not permit any of these conditions to occur.

3.2 Software/OS Consistency

Software issues can present a larger challenge to dynamic system reconfiguration. While the hardware built into standard microprocessor devices protects against low-level hardware conflicts, there are many more details that must be managed. Figure 2 below summarizes some of the potential problems from an improper reconfiguration.

The example shows an initial configuration of 3 processes (A, B, and C) in the normal operational state. A reconfiguration occurs, changing to a new configuration. The new configuration replaces these process A with A', C with C' and removes process B altogether. The bottom half of the figure shows the new configuration, along with the potential errors.

1. The memory associated with process A was not needed in the subsequent configuration, but was not returned to the free pool. The result is a memory leak. Long-term reliability will be adversely affected.
2. Process B's task structure was not recovered, resulting in an orphan task within the real-time schedule. The result of this error can be extra tasks executed by the kernel, with a loss in performance.
3. During the reconfiguration event, a message was in transit between processors. By the time the message reaches its destination, the receiving process no longer exists, and possibly, the OS/kernel drivers have been reconfigured. The communication channel is rendered inoperative.

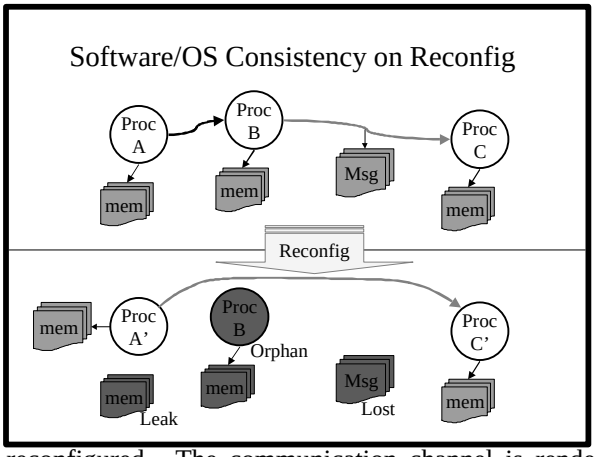

Figure 2: Software/OS Reconfiguration: Maintaining Consistency

3.3 Application-Level Consistency

At a higher level, the application's requirements and implementation details impose restrictions in the reconfiguration process. Typically, these attributes are highly application-specific. Two examples of consistency requirements are displayed in Figure 3 below.

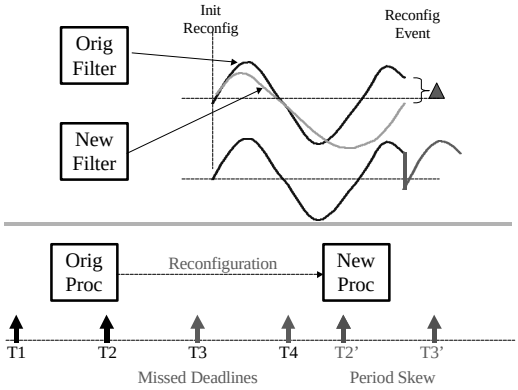

Figure 3: Maintaining Application Consistency Through Reconfiguration

When a signal processing or controls application produces outputs, certain signal qualities must be preserved. Often an external system will require signal output continuity and/or continuous first derivative properties. In the example, which swaps filters online, the new filter is operating out of sync with the original filter. A rapid switchover will create a discontinuity in both the signal and its first derivative.

Another problem occurs in the attempt to maintain real-time constraints during reconfiguration. If the reconfiguration cannot be completed in sufficient time, deadlines will be sacrificed. In addition, the timebase can be shifted, resulting in a skew in system output period.

4. Reconfiguration Strategies

From the previous description of potential problems, it is clear that reconfiguration support must be built into the design approach, from the lowest levels of the execution environment, to the high-level design/requirements capture tools. The extent of support is defined by the requirements of the target systems. The driving factors include how fast the system must reconfigure, whether intermediate states must be preserved (Application Signal Continuity), and if timing must be preserved. In this section, we examine the potential reconfiguration strategies and their impact on system capabilities.

4.1 Reboot Strategy

The first reconfiguration strategy is termed the "Reboot" approach. This is the simplest approach. It involves the orderly shutdown of tasks, bringing the system to a known, clean state. From this state, a new processing structure is constructed. The implementation for this approach is simple, requiring the minimum amount of non-standard support from the execution environment. Since there is no overlap in execution of processes in adjacent states, there is no need for additional processing capability.

The drawbacks of this approach are severe. The system is offline during the reconfiguration time. No events can be handled, so a system under control is open-loop during that time. There is no provision for preservation of state. This can lead to long recovery times when the new configuration is started. Both of these factors lead to system application transients, both timing and signal continuity. This approach is not suited for the majority of embedded, closed-loop systems.

4.2 State Transition Approach

The second approach allows the insertion of transitory states between the major system operating modes. These states allow the system to take smaller steps between operational modes, resulting in smaller transients. The intermediate configurations inherit state from their predecessors. The intermediate algorithms must be designed to gradually turn the system around. While not continuous, the steps can be made arbitrarily small.

This approach has several positive aspects. The state preservation allows transients to be minimized. The magnitude of the steps can be chosen by the designer to minimize key application behaviors. Few spare resources are needed, since the system is operating in only one mode at a time. The flexibility is limited only by the designers and by the time available for the transition.

There are several difficulties in this approach. The execution infrastructure must support the rapid transition of processes. The environment must avoid all of the potential hazards discussed previously. The state of the processes must be retained from the prior step, mapped to the structures required by the next step, and installed with the new processing structure. For software, this involves copying of memory, with possible reformatting of structures. For hardware, registers must be initialized. The computation of the mapping may be complex.

The design of intermediate states can be complex, depending on the application. These transitory states depend both on the initial state and the final state, the algorithm characteristics, and the timing requirements. For smooth application transitions, many intermediate states may be required, leading to long transition times. (It should be noted that the application system is still under control during transition, but probably not the optimal algorithm).

4.3 Parallel State Transition Approach

An extension of the State Transition approach allows the system to execute several modes in parallel. This has the same benefits as the state transition approach with the added benefit of being able to execute algorithms prior to use, in an offline mode. The state of the offline process can be allowed to stabilize prior to impacting upon system performance. When transients have disappeared, the system can be transitioned to the new state.

This approach has several benefits. The application-level transients can be minimized by proper design. The downtime is minimal, as is the operation of the system in a less-than-optimal configuration. Multiple states can be preserved, not forcing all information to be encoded in one format. This minimizes the impact of the design of one mode on another, thus simplifying design.

There are also several drawbacks. The underlying runtime environment must support mechanisms for rapid stepping between processes, the ability to execute multiple threads simultaneously, and the combination of attributes from the parallel executing processes.

System design is complicated by the need to design parallel structures. (In some cases, the parallel approach allows design separability, simplifying matters.) The

necessary computational resources are increased, due to the need to execute multiple parallel processes.

Given the difficulties of implementation, the capabilities of this approach are required to service many reconfigurable application domains.

5. Execution Environment Design

The previous sections lay out a set of requirements for the capabilities of the execution environment. They also point out some of the design complexities. In a vacuum, the execution environment cannot solve these problems. The overall system design approach must span from the top-level algorithm designers/system requirement & resource specifications down to the hardware/software implementations. The top-level design issues have been discussed in terms of a domain-specific modeling environment, where the environment is tuned to reconfigurable system design. The Execution Environment forms the infrastructure onto which these designs are projected.

The Execution Environment must be designed with an interface suitable for synthesis from a MIC-Generator approach. The concepts, properties and interfaces of the runtime environment must be compatible with the design representation and synthesis approach. Capabilities and interfaces should be tuned to simplify the generator. This requirement demands a simple, uniform interface with a well-defined, consistent set of semantics that apply throughout the system. Since the system includes software, hardware, and interactions between parallel modules, a common structure must map to a wide range of components.

The execution environment concepts have been driven by results from using tools developed over the past several years. These tools are currently used to construct large-scale, parallel, real-time signal processing systems. The runtime environment enabled development of CADDMAS systems, which are used by the USAF for turbine engine testing and NASA for SSME monitoring and analysis [4].

The semantics of the execution environment implement a large-grain dataflow architecture. The Worker Function captures the tasks that are performed by the system. Communication nodes capture the transfer of data between workers. Computations can be described as a bipartite graph, where workers connect to Comm nodes, and Comm nodes connect to workers. At this level, there are no implied semantics of the workers. The execution properties of workers (Data tokens produced/consumed per execution, timing of execution, etc) are maintained at a higher level. The semantics of the Comm units are asynchronous queues.

When the generic large-grain dataflow graphs are implemented, they must be mapped down to a physical implementation. The implementation takes the form of either software or hardware. Software workers execute on a DSP or CPU, which we term Processes. Hardware workers are either implemented in reconfigurable hardware (FPGA's), ASIC implementations, or combinations of both. Processes and Processors are logically equivalent, representing functions on data. Processes/Processors are connected via logical Comm that must buffer, communicate, and match data formats. In software implementations, the Comm object is implemented by the OS/Kernel as a Stream, a software queue in memory. In hardware, the Comm object is implemented with registers and/or FIFO, or simply wires (Figure 4).

The execution environment spans software and reconfigurable hardware. The software environment consists of a simple, portable real-time kernel with a run-time-configurable process schedules, communication schedule, and memory management [5]. Communications interfaces are supported within the kernel, making cross-processor connections invisible. Memory management is integrated with the scheduler and communication subsystems, enabling (but not solving) the problems associated with dynamic reconfiguration. The kernel allows dynamic editing of the process table, and of the communications maps. The proper sequencing of these operations, including task execution phases, is necessary for the avoidance of reconfiguration problems. The current approach supports the "Reboot" approach directly, and will support the more advanced reconfiguration approaches with cooperation of the application tasks.

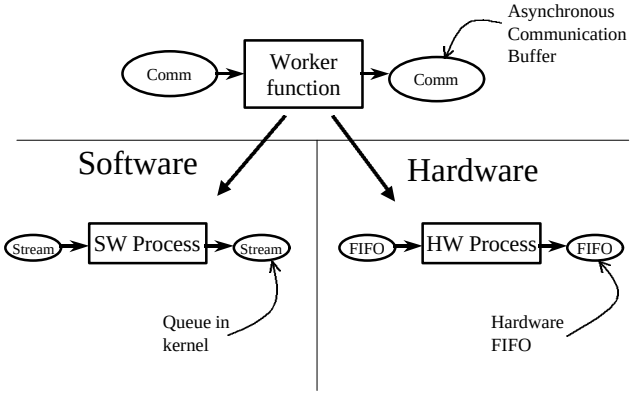

Figure 4: Common Execution Semantics

The hardware execution environment is semantically similar. The implementation, however, is much different. The Virtual Hardware Kernel exists as a concept used in the system synthesis. The MIC Generator synthesizes a set of VHDL structural codes, one for each configurable device multiplied by the number of operational modes. Processors are directly synthesized using predefined components. Communications elements are selected from a library of interface types, based on the requirements of the workers on either end, the required performance, and the available resources. The communication infrastructure works in cooperation with the software communications, performing the signal buffering, and the necessary off-chip interfaces and data converters.

6. System Synthesis

The multi-aspect model of the system describes a possibly enormous number of design solutions. The set design solutions must be evaluated to find a set of designs (mode configurations) that best satisfy a number of design criteria. This is a very difficult task because there are inherently a large number of conflicting design criteria in reconfigurable systems. Each mode has performance requirements that demand a certain level of performance from the hardware for a given algorithm. Some hardware components are more suitable for certain tasks than others. DSPs provide a general-purpose solution and reasonable performance for many complex algorithms while ASICs can provide a high-performance solution at the cost of adding a dedicated fixed-function IC. The processing needs of the multiple modes of the system must be met for a single shared hardware platform. The synthesis process will select feasible hardware/software architectures for user-definable cost functions such as weight, power, algorithmic accuracy and flexibility.

The structural aspect that captures the hierarchical data-flow with alternatives can represent an exponentially large design space in a compact form. An algorithm structure may have an extremely large number of possible implementations. Searching the design space for a set of configurations to satisfy the design criteria is accomplished in multiple stages, each with increased resolution. A symbolic constraint satisfaction method is introduced to provide an initial pruning of the design space. This method operates on the binary design and performance constraints specified for the systems. This method of pruning is implemented using a symbolic method known as ordered binary decision diagrams (OBDD). Information represented in the form of a Boolean function can be represented efficiently in the form of an OBDD [6]. A symbolic representation is built along with a constraint set. System design alternatives that do not satisfy this constraint set may be quickly eliminated. Through the use of the OBDD representation these design possibilities do not have to be examined individually allowing for an extremely large design space to be quickly narrowed.

The design search will continue to narrow down possibilities through high-level simulation of the system. Components will have associated performance models that can be used to compute performance data of the system configuration being evaluated such as communication utilization, processor utilization, etc. Finally when the searching process has narrowed the design space down to only a few candidate configurations it may be necessary to fully synthesize these configurations for the target platform and evaluate their performance through actual run-time measurements.

At this point, the synthesis procedure can generate the actual runtime artifacts. From the behavioral models, a set of tables is produced for the Configuration Manager. This defines the behavior of the system, in terms of a state machine. These tables are executed directly by the configuration manager.

For each configurable component, a set of design files is generated. One file is built for each component for each mode. The design is specified in structural VHDL, using computational components from the design library and interface components from the runtime system library. These VHDL files are then compiled using vendor-supplied/COTS VHDL compilers and part-specific Place-and-Route tools. The result is a “bitfile”, ready for direct device programming.

For the general-purpose/DSP components, a set of real-time schedule specifications and communication maps are generated. These are then processed into a set of object modules and tables for direct download into the parallel array of processors. One set of tables is built for each processor for each mode, and one common executable module is generated for each processor.

The result of the synthesis and post processing is a complete executable system, ready for deployment. Current synthesis capabilities do little optimization (optimization is an ongoing research topic).

7. Conclusions

Certain high-performance, highly-constrained applications need to be adaptable to their requirements and environment. Needed is a method for rapid, automated system synthesis that can provide maximal use of available hardware over time through system reconfiguration. FPGAs are the enabling technology for a computing platform that is able to adapt to changes in the processing algorithm. FPGAs may be used as general purpose computing devices whose flexibility and speed fall somewhere between that of a custom ASIC and that of a standard CPU.

The use of a domain-specific, multi-aspect modeling paradigm is key to capturing all relevant information about the system in an integrated environment. With this information design choices can be automated to select from the possible configurations a system configuration that meets specified design requirements and is also optimized for other specified metrics such as power use, weight, and algorithmic accuracy. Extremely large design spaces can result from the freedom given in defining the algorithm structure alternatives. Methods are being investigated to manage these large state spaces symbolically using Ordered Binary Decision Diagrams (OBDDs). Symbolic manipulation can provide a way to prune the design space without examining each design alternative individually.

The current modeling environment is being evaluated. Applications are being modeled to ensure all relevant information is captured in the models. A set of intrinsic components is being constructed for the hardware communication interfaces. Currently, systems are being synthesized for direct hardware implementation from the models. Dynamic reconfiguration approaches are being tested and refined.

Many issues need to be explored with respect to dynamic reconfiguration at the application level. Suppose processing algorithm A is operating and reconfiguration occurs to switch over to algorithm B, what does an output common to both of these algorithms? Is there a discontinuity at the point of reconfiguration observed? If these outputs are used to provide information to a control system this may be unacceptable.

References

[1] J. Sztipanovits, G. Karsai, T. Bapty, “Self-Adaptive Software for Signal Processing,” *CACM*, Vol. 41, No. 5, pp. 66-73, May, 1998.

[2] J. Scott, T. Bapty, S. Neema, J. Sztipanovits, “Model-Integrated Environment for Adaptive Computing,” MAPLD '98, *Proceedings of the 1998 Military and Aerospace Applications of Programmable Devices and Technologies Conference*, Sept 15-16, 1998, Greenbelt, MD.

[3] J. Sztipanovits, et al., “MULTIGRAPH: An Architecture for Model-Integrated Computing” *Proceedings of the IEEE ICECCS’95*, Ft. Lauderdale, Florida, Nov. 6-10, 1995.

[4] B. Abbott, T. Bapty, C. Biegl, G. Karsai, J. Sztipanovits, "Model-Based Software Synthesis," *IEEE Software*, pp. 42-53, May, 1993.

[5] Bapty, T., Abbott, B., "Portable Kernel for High-Level Synthesis of Complex DSP-Systems," *Proceedings of ICSPAT '95*, Boston, MA, Oct 24-26, 1995

[6] R. Bryant, “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams,” Technical Report: CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, 1992.