A Design Notation and Toolset for High-Performance Embedded Systems Development

Devesh Bhatt and John Shackleton

Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418. USA

Abstract. In traditional design methodologies, the system designer typically develops the application in a sequential paradigm almost to completion before addressing issues of parallelism and mapping to a heterogeneous architecture. As the architectural complexity of these applications increase, however, this process becomes too costly since implementation must be started anew after the design. The quality of the design also often suffers as a result. This is especially true for embedded applications, where the complexity lies within the system software and hardware architecture. We present a new methodology and toolset aimed at improving the system development process for high-performance embedded applications. The toolset provides a unified design representation from early design specification to integration—allowing for parallelism and synchronization specification in domain specific styles, and automating many process steps such as partitioning/mapping, simulation, glue-code generation, and performance analysis.

1 Introduction

The increased availability of relatively inexpensive embedded architectures has made it feasible to implement more and more high-performance applications. The potential benefit of parallel architectures, however, is often offset by the effort needed to develop and port applications on these architectures. This effort is further increased for embedded systems due to their real-time requirements and due to the complexity of integrating interacting application functions that may use different styles of parallelization and synchronization. This is unlike typical scientific applications that implement a single central algorithm. In addition, interactions between components must be understandable and verifiable for mission-critical applications.

In spite of recent progress in object-oriented design methods, parallel languages, communication libraries, and real-time operating systems, substantial manual effort is needed in developing an application using these often diverse technology components. The system designer typically develops the application in a sequential paradigm almost to completion before addressing issues of parallelism and mapping to a multilevel heterogeneous architecture, and using the communication and operating system services. Thus, the implementation becomes disjoint from the design—resulting in duplication of effort and inconsistencies.

The Multi-Domain Embedded System Architect (MESA) is a methodology and toolset that bridges this gap and addresses many issues facing the developers of complex embedded systems. MESA provides the following capabilities:

- an end-to-end development process driven by a unified design notation

- a design notation for specification of parallelism, synchronization, and scheduling properties within domain-specific programming styles.
- automated partitioning, mapping, analysis, and simulation of the complete software and hardware design.
- automated glue-code generation to specific target communication and operating system services on the hardware architecture, with performance monitoring

This paper is divided into five sections. Section 2 discusses the current design methods and languages and the need for domain-specific approaches. Section 3 presents an overview of the MESA methodology, focusing on the design notation. Section 4 presents some features of the MESA toolset in the context of an application. Finally, Section 5 presents the current MESA status and future plans.

2 Background and Motivation

2.1 Object-Oriented Methodologies

Much progress has been made recently in object-oriented (OO) design methods and tools for applications on sequential platforms. Published OO methodologies, such as Coad Yourdon [2], Shlaer-Mellor [3], and OMT [4], and their implementation in commercial Computer Aided Software Engineering (CASE) tools is gaining widespread acceptance in many application domains; typically those domains that do not require high-performance architectures and real-time operation.

Information Complexity vs. Architectural Complexity. In a typical system development today, one encounters two kinds of complexities:

1. *Information Complexity.* This is due to the different types of data, objects, inheritance, and relationships. The class diagrams in OO methodologies allow specification of these aspects for information-intensive applications such as databases.
2. *Architectural Complexity.* This is due to the parallelization/distribution of software functions, data striping, data buffering and pipelining, mapping of functions onto hardware, scheduling to meet real-time requirements. Embedded applications such as radar signal processing, tracking, automatic target recognition, avionics mission management, industrial process control have this kind of architectural complexity.

For example, a typical PC database application might exhibit 95% information complexity and 5% architectural complexity. On the other end, a typical embedded signal processing application might exhibit 5% information complexity and 95% architectural complexity. While the OO methodologies do a good job of analyzing and designing information-intensive applications, they are inadequate for a large class of embedded applications, especially those utilizing complex architectures to achieve high performance. We have indeed found this to be the case in our development of signal processing and tracking applications.

2.2 Parallel Languages and Models

To address certain issues of architectural complexity for parallel and distributed architectures, much progress has been made lately in languages, middleware, and operating systems. This includes: programming models such as Actors; programming