A FRAMEWORK FOR OBJECT-ORIENTED EMBEDDED SYSTEM DEVELOPMENT BASED ON OO-ASIPS

NASER MOHAMMADZADEH* and SHAHIN HESSABI†
Department of Computer Engineering,
Sharif University of Technology,
Azadi Ave., Tehran, Iran
*mohammadzadeh@aut.ac.ir
†hessabi@sharif.edu

MAZIAR GOUDARZI
System LSI Research Center, Kyushu University,
Fukuoka, Japan
goudarzi@sbrc.kyushu-u.ac.jp

MAHDI MALAKI
Department of Computer Engineering,
Sharif University of Technology,
Azadi Ave, Tehran, Iran
maleki@ce.sharif.edu

Revised 15 April 2008

The growing complexity of today’s embedded systems demands new methodologies and tools to manage the problems of analysis, design, implementation, and validation of complex-embedded systems. Focusing on this issue, this paper describes a design and implementation toolset using our ODYSSEY methodology, which advocates object-oriented (OO) modeling of embedded systems and its ASIP-based implementation. The proposed approach promotes a smooth transition from high-level object-oriented specification to the final embedded system, which is composed of hardware and software components. The transition from higher to lower abstraction levels is facilitated by the use of our GUI, which supports the intermediate steps of the design and implementation process. In order to illustrate the proposed approach and related toolset, we apply this top-down design and implementation framework to real-world embedded systems, namely JPEG codec and Motion JPEG codec. Experimental results show that the developed tool remarkably decreases the design and verification time with modest performance penalty.

Keywords: Embedded systems; object-oriented design; design automation tool; polymorphism; Application-Specific Instruction Processor (ASIP).
1. Introduction

Embedded systems are usually implemented as combinations of hardware with general purpose computational capabilities and more dedicated modules. Together, they perform a function carefully partitioned in software and hardware to obtain the optimum trade-off between the various quality metrics. This type of implementations has become increasingly popular as advances in integrated-circuit technology and processor architectures allow flexible computational parts and high-performance modules integrated on a single chip.

Despite advances in manufacturing technology, design-technology lags far behind; it is nowadays a common practice that all system-level decisions are taken ad hoc in the beginning of the design process. Months of efforts are then invested in realizing them, often manually hand-crafting the embedded software in assembly language. Since there is usually no time left for the second try, many of these decisions have to be conservative to guarantee the system correctness. Also, the design reuse is limited, which results in longer time-to-market times. Clearly, this leads to less market-competitive products.

The design automation for embedded systems is becoming a “must”. What is needed is an interactive environment that supports the designer, not only in transforming a high-level specification into a suitable implementation, but also in reusing the already implemented applications. The environment should allow fast experimentation with different architectural options and relieve the designer from the burden of more time-consuming, but often simple, synthesis tasks.¹ ²

FPGA and programmable SoC manufacturers offer their own tools to ease the implementation of hardware–software systems on them. Examples include Platform Studio³ and FastChip³ from Xilinx. Although such tools simplify the design process, the designer still needs to manually design individual hardware and software parts and assemble them together using the tool facilities. This makes the design process very time-consuming and error-prone, and hence, very hard and expensive to make changes to the partitioning later in the design process.

Software-based approaches to hardware design have recently received more attention. DK Suite⁴ from Celoxica enables the designer to implement hardware from an extended-C source code. Similarly, Catapult⁵ from Mentor accepts C/C++ source code. However, none of these tools extend to synthesize a hardware–software system. Moreover, they do not support object-oriented polymorphism.

Other tools, such as VCC⁶ and SPW⁶ from Cadence and CoCentric System Studio⁷ from Synopsys, exist that accept finite-state machine and/or graphical design entry and produce hardware–software systems. System design using such tools is not generally as easy as programming in C/C++, and hence, such tools are not likely to have the effect that we wish to see in the popularity of hardware and hardware–software designs among beginners.

Focusing on this issue, in this paper we introduce an object-oriented platform-based design process using our ODYSSEY⁸ design methodology that advocates designing an “application-specific instruction-set processor” (ASIP) tailored to the
application-domain being supported; this ASIP can then be programmed to add missing functionalities and/or to correct malfunctioning parts. The approach is based on the reuse of hardware and software components and on the configuration of FPGA-based architectural platforms. Our approach aims to ensure a smooth transition from object-oriented models specified in C++ to the target embedded system. The transition from higher to lower abstraction levels is facilitated by a real-time J# GUI.  

The remainder of this paper is organized as follows. In Sec. 2, we mention a motivational example and related works. Section 3 reviews the ODYSSEY design methodology. In Sec. 4, we describe our synthesis flow and the toolset. The structure of system hardware is described in Sec. 5. Section 6 contains case studies implemented by our approach. Finally, we conclude in Sec. 7.

2. Motivational Examples and Related Works
   
   2.1. Examples to motivate the methodology and the tool
   
   Suppose we want to implement a JPEG decoder as a hardware–software co-designed implementation. The first step is to describe our desired system with a high-level language such as C/C++, and debug and verify the system functionality. The second step is the partitioning of methods into hardware and software, and designing a protocol to connect these parts. In the next step, software designers design software parts, and hardware designers design hardware modules. So far, everything sounds good. The last step is integration and debugging. In this stage, problem begins to appear: how is overall system verified and debugged? Integration of such system seems to be one of the main challenges. Lack of a well-defined design flow is another problem in designing HW/SW systems. So, if a methodology and tool can generate a HW/SW system from its high-level description and provide debugging and verifying facilities, integration will no longer be a problem. The ODYSSEY does this work. The ODYSSEY is a methodology and a tool that provides a framework to design, debug, and verify such systems. Moreover, reuse is another important feature supported by ODYSSEY. For example, to design a Motion-JPEG decoder from scratch, all methods should be developed and debugged, but if a JPEG library exists, and it is possible to use this library, the design time intensively decreases. Our tool spans from object-oriented model of application to low HDL model. In this paper, we present our methodology and framework for the development of embedded systems based on OO-ASIPs.

   2.2. Related work
   
   A large body of research is available that can be related to this research in one direction or another. Table 1 lists a number of such works. Vulcan, 11 Cosyma, 12 SpecSyn, 13 Lycos, 14 and Polis 15 have conducted research on hardware–software partitioning. CoWare 16 and Chinook 17 have worked on hardware–software interfacing. Cosyn 18 and SOS 19 have focused on hardware–software co-synthesis. Cosmos 20
Table 1. A number of related works in the area of hardware–software co-design and system-level design.

<table>
<thead>
<tr>
<th>Project</th>
<th>University</th>
<th>Main focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinook</td>
<td>U Washington</td>
<td>Interfacing</td>
</tr>
<tr>
<td>Cobra</td>
<td>U Tubingen</td>
<td>Prototyping</td>
</tr>
<tr>
<td>Cosmos</td>
<td>TIMA</td>
<td>Refinement</td>
</tr>
<tr>
<td>Cosyama</td>
<td>TU Braunschweig</td>
<td>Partitioning</td>
</tr>
<tr>
<td>Cosyn</td>
<td>Princeton</td>
<td>Cosynthesis</td>
</tr>
<tr>
<td>Coware</td>
<td>IMEC</td>
<td>Interfacing</td>
</tr>
<tr>
<td>Javatime</td>
<td>UC Berkeley</td>
<td>Refinement</td>
</tr>
<tr>
<td>Lycos</td>
<td>TU Denmark</td>
<td>Partitioning</td>
</tr>
<tr>
<td>Polis</td>
<td>UC Berkeley</td>
<td>Partitioning, Verification</td>
</tr>
<tr>
<td>Ptolemy</td>
<td>UC Berkeley</td>
<td>Modeling, Simulation</td>
</tr>
<tr>
<td>SOS</td>
<td>U Southern California</td>
<td>Cosynthesis</td>
</tr>
<tr>
<td>SpecSyn</td>
<td>UC Irvine</td>
<td>Specification, Partitioning</td>
</tr>
<tr>
<td>Vulcan</td>
<td>Stanford</td>
<td>Partitioning</td>
</tr>
</tbody>
</table>

and Javatime\textsuperscript{21} proposed sets of refinements to gradually implement a hardware–software system from a given specification. Cobra\textsuperscript{22} concentrates on prototyping and Ptolemy\textsuperscript{23} focuses on modeling and simulation of heterogeneous systems. As Table 1 clearly shows, there are several issues and concerns in system-level design that make it impractical and unreasonable, to consider all of them at the same time. This has made all researchers in the field to focus merely on one, or a few, of such issues, and we are not an exception.

Our ESL design methodology is developed on two points: object-orientation in modeling and ASIP in implementation. These same choices of starting and ending points in the design of hardware–software embedded systems comprise the main difference between our methodology and all other related works; they either do not start from an OO application or do not implement it as an ASIP. This research focuses on the synthesis of an ASIP (its hardware, firmware, and their interface) and on generating the software to run on that ASIP such that the ASIP that is developed for a given set of applications can equally well serve future applications that reside in the same domain. Specifically, regarding partitioning (i.e., the choice of parts to implement in hardware when the ASIP is first synthesized), ODYSSEY toolset enables the designer to automatically realize any desired partitioning choice.

Four major works exist that directly relate to our methodology: the OASE project,\textsuperscript{24} the ODETTE project,\textsuperscript{25} the Silicon Infusion’s Enodia\textsuperscript{R} architecture, and work done by Cheng and Wu.\textsuperscript{27} A summary of their main features is given in Table 2 to facilitate easier comparison. ODETTE and OASE synthesize ASIC from the OO models and consequently do not provide support for application extensions in future. This contradicts a major feature of OO methodology, i.e., extensibility and reusability. Enodia\textsuperscript{(R)} has many things in common with our approach, but follows a bottom-up design flow rather than our top-down flow for application development. In Ref. 27, Cheng and Wu design and implement software objects in
Table 2. Comparison between ODYSSEY and the related work.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>OASE</th>
<th>ODETTE</th>
<th>Enodia®</th>
<th>Cheng’s and Wu’s work</th>
<th>ODYSSEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling language</td>
<td>Java, SystemC</td>
<td>Objective-VHDL SystemC-Plus</td>
<td>Not available</td>
<td>Java</td>
<td>C++</td>
</tr>
<tr>
<td>Implementation style</td>
<td>ASIC</td>
<td>ASIC</td>
<td>Heterogeneous multiprocessor</td>
<td>FPGA</td>
<td>Uniprocessor ASIP</td>
</tr>
<tr>
<td>Object synthesis</td>
<td>Analysis and optimization</td>
<td>Per-object method replication</td>
<td>Multiple objects per method implementation</td>
<td>—</td>
<td>Multiple objects per method implementation</td>
</tr>
<tr>
<td>Optimizations</td>
<td>Objects reachability</td>
<td>Dead-code removal</td>
<td>Not available</td>
<td>Not available</td>
<td>—</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Method inlining</td>
<td>Method replication and multiplexing</td>
<td>Firmware</td>
<td>Not supported</td>
<td>Overhead-free (by routing packets in NoC)</td>
</tr>
<tr>
<td>HW-SW implementation</td>
<td>Stub generation</td>
<td>Not provided</td>
<td>Software on multiprocessor</td>
<td>Not provided</td>
<td>Software on uniprocessor OO-ASIP</td>
</tr>
<tr>
<td>Model of concurrency</td>
<td>Multiple processes in modules</td>
<td>Objects involved from processes</td>
<td>Not available</td>
<td>Not available</td>
<td>Inside method implementations</td>
</tr>
<tr>
<td>Dynamic (de)allocation</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Supported</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>
They translate Java classes into VHDL codes by SOCAD. They do not support polymorphism that is one of our main contributions.

3. The ODYSSEY System-Level Design Methodology

Software accounts for 80% of the development cost in today’s embedded systems, and object-oriented design methodology is a well-established methodology for reuse and complexity management in the software design community. These facts motivated us to follow top-down design of embedded systems starting from an object-oriented embedded application. The OO methodology is inherently developed to support incremental evolution of applications by adding new features to (or updating) previous ones. Similarly, embedded systems generally follow an incremental evolution (as opposed to sudden revolution) paradigm, since this is what the customers normally demand. Consequently, we believe that OO methodology is a suitable choice for modeling embedded applications, and hence, we follow this path in ODYSSEY.

The other fundamental choice in ODYSSEY is the implementation style. ODYSSEY advocates programmable platform or ASIP-based approach, as opposed to full-custom or ASIC-based philosophy of design, since the design and manufacturing of full-custom chips in today’s 60 nm technologies and beyond are so expensive and risky that increasing the production volume is inevitable to reduce the unit cost. Programmability, and hence programmable platform, is one way to achieve higher volumes by enabling the same chip to be reused in several related applications, or different generations of the same product. Moreover, programming in software is generally a much easier task compared to design and debug of a working hardware. Therefore, programmable platforms not only reduce design risk, but also result in shorter time-to-market.

ODYSSEY synthesis methodology starts from an object-oriented model and provides algorithms to synthesize the model into an ASIP and the software running on it. The synthesized ASIP corresponds to the class library used in the object-oriented model, and hence, can serve other (and future) applications that use the same class library. This is an important advantage over other ASIP-based synthesis approaches, since they merely consider a set of given applications and do not directly involve themselves with future ones.

One key point in the ODYSSEY ASIP is the choice of the instruction-set: methods of the class library that is used in the embedded application constitute the ASIP instruction-set. The other key point is that each instruction can be dispatched either to a hardware unit (as any traditional processor) or to a software routine; consequently, an ASIP instruction is the quantum of hardware–software partitioning. Moreover, it shows that the ASIP internals consist of a traditional processor core (to execute software routines) along with a bunch of hardware units (to implement in-hardware instructions as shown in Fig. 1).
Fig. 1. Internal architecture of an OO-ASIP corresponding to a class A with \( f() \) and \( g() \) methods, and class B derived from A while redefining \( f() \) and \( g() \) and introducing \( h() \) method.

An OO application consists of a class library, which defines the types of objects and the operations provided by them, along with some object instantiations and the sequence(s) of method calls among them. We implement methods of that class library as the ASIP instructions, and realize the object instantiations and the sequence of method calls as the software running on the ASIP. A simple internal architecture for such an ASIP is presented in Ref. 8 and summarized in Sec. 3.1.

### 3.1. ASIP architecture

A simple internal architecture of the OO-ASIP is shown in Fig. 1. It corresponds to a library comprising two classes, A and B, where B is derived from A and has overridden its \( f() \) and \( g() \) methods and has introduced an \( h() \) method. The following C++-like code excerpt demonstrates this. Note that redefinitions of the same method can reside in different partitions (e.g., \( A :: g() \) is a software method while \( B :: g() \) is a hardware one).

```cpp
class A {
    void f();
    void g();
    ...; // other member-functions and attributes
};
class B extends A {
    void f(); // A::f() is overridden here
    void g(); // A::g() is overridden here
    void h();
    ...; // other member-functions and attributes
};
```
All objects’ data are stored in a central data memory accessible through an Object Management Unit (OMU). In the application corresponding to Fig. 1, three objects are defined: $O_{A1}$, $O_{A2}$, and $O_{B1}$. Objects of the same class (e.g., $O_{A1}$ and $O_{A2}$) have the same layout and size in memory for their attributes. Objects of a derived class keep the original layout for their inherited attributes (the white part of the memory portion of $O_{B1}$) and append it with their newly introduced attributes (the gray part of $O_{B1}$ box in Fig. 1).

The class methods that are assigned to hardware partition, i.e., the hardware methods, are implemented as Functional Units (FU); the other class methods, i.e., the software methods, are software routines stored in the local memory of the traditional processor core (the upper-left box inside the OO ASIP in Fig. 1).

4. Our Synthesis Flow and Environment

Our ESL design environment is a system-level design tool for design and implementation of embedded systems based on OO-ASIPs. The tool includes an interactive graphical tool for designing, simulating, and debugging in different abstraction levels. It takes, as input, a set of header and program files that together define the class library as well as the main() function, where the objects are instantiated and the sequence of method calls among them is specified, and produces, as output, an OO-ASIP-based architecture. The transition from object-oriented model to the final OO-ASIP-based architecture walks through intermediate layers. The generated system in each layer can be simulated and debugged. Figure 2 shows our synthesis flow for the development of an OO-ASIP for an application from scratch. If an application is an extended version of an already implemented OO-ASIP, where extension can be implemented by only adding software methods, some of the intermediate steps are deleted. The only required steps for extending OO-ASIP are to synthesize the object-oriented version of extended application by ODYSSEY system-level synthesizer (see Fig. 2) and then updating the final bitstream.

The entire process is divided into two layers: the upper layer, named system-level synthesis, takes the system model in C++ and produces the software and hardware architectures in a mixture of structural and behavioral modeling styles. This synthesizer is depicted in the figure as ODYSSEY System-level synthesizer. The resultant system from this process is a high-level OO-ASIP co-simulation model whose software methods are in C++, and hardware methods are in systemC. This co-simulation model can be simulated by Modelsim or compiled and run by a C++ compiler. In this level, the software methods are run on a systemC Module that plays the traditional processor’s role.

*Modelsim 6.0 (and later) supports systemC. Modelsim is a registered trademark of Mentor Graphics Corporation.*
Fig. 2. Our design and implementation flow.

The lower layer, called *downstream synthesis*, takes the above-generated hardware and software partitions and produces gate-level hardware and object code software.

In the first step of this layer, the hardware methods described in SystemC are synthesized by Synopsys SystemC Compiler® and translated into VHDL. The tool connects to a Synopsys server whose IP address is set when a project is developed and sends SystemC model of hardware methods to the server and receives the resultant VHDL model of functional units.

To elevate the simulation and verification speed in this layer, a cycle-accurate co-simulation environment is developed. This environment is composed of a processor Instruction Set Simulator (ISS) integrated with a hardware simulator, and communicate with it through socket connections. As shown in Fig. 2, we call this level as a system ISS model. Figure 3 shows the co-simulation environment structure.\(^3\)

\(^{3}\)We call hardware methods “functional units”.

\(^{b}\)We call hardware methods “functional units”.
In the final layer, the ISS model of a processor is replaced with MicroBlaze\textsuperscript{10, c} HDL code. At this level, system can be simulated by Modelsim or emulated by ML410 development board.\textsuperscript{3} The system software methods are compiled and run on a processor.

As Fig. 2 shows, the tool also has an IP core support. In other words, the user can use pre-designed IP cores instead of top-down behavioral synthesis for some functional units. Simulation models of the imported IPs are put in system-level simulation. Imported IPs are not synthesized during the system-level synthesis process and are replaced with their HDL models during the downstream synthesis process.\textsuperscript{34}

The tool’s design and implementation environment is shown in Fig. 4. Its graphical user interface (GUI) consists of various windows that give access to parts of the design. The primary access point in the GUI is the Main window. The Main window provides convenient access to design libraries and objects, source files, debugging commands, simulation status messages, etc. Other windows and their functions are summarized as follows:

- **Project Explorer:** Workspace tabs organize design elements in a hierarchical tree structure.
- **Transcript:** The Transcript pane reports status and messages generated in each command run.
- **Multiple Document Interface (MDI) Pane:** The MDI frame is an area in the Main window where source editor is displayed. The frame allows multiple windows to be displayed simultaneously.
- **Process View:** The process view window gives access to processes that can be run on each level.

As Fig. 4 shows, processes are categorized into three main groups: Synthesis, Simulation, and Program Device. Synthesis part contains four processes: first process, “Generate Co-simulation Model”, generates the systemC co-simulation model of

\textsuperscript{c}MicroBlaze is a registered trademark of Xilinx, Inc.
A Framework for Object-Oriented Embedded System Development

Fig. 4. Our tool’s graphic user interface.

the system; second process, “Generate Hardware Model”, generates VHDL model of hardware methods; third process, “Generate EDK Model”, compiles the system software methods and produces the final Simulation HDL Model; the final process, “Generate Programming Files”, synthesizes the system’s HDL model and generates the final bitstream that can be downloaded on FPGA. Second process category, “Simulation”, is composed of four processes: the first process, “Pre-synthesis Model”, runs the object-oriented model of the system; the second process, “Co-simulation Model”, simulates the SystemC co-simulation model of the system with Modelsim; the third process, “ISS Model”, simulates the HDL model of the system on our cycle-accurate co-simulation environment; the final process of this category, “Post-synthesis HDL Model”, simulates the final system HDL model with Modelsim. Program Device category of processes is to download the bitstream on an FPGA platform.

5. The Structure of Final System

The resultant OO-ASIP synthesizable system is composed of four main parts: Processor, OMU, FUs, and packet network, as shown in Fig. 5.

The software part of our design runs on a MicroBlaze. MicroBlaze is a 32-bit RISC processor which runs with a 100MHz clock. The processor connects to OMU and packet networks through two FSL (fast simple link) peripherals.
The OMU module consists of OMU network, object memory, and mapper. The object data is stored in the OO-ASIP main memory. We call this memory “object memory”. The OMU network is responsible for synchronizing the Processor and FUs access to the objects’ data. The Processor or requesting FU provides the OMU with the address of the requested data. The address consists of the oid (object id) of the object and the index of the data item within the object data storage. The oid is composed of onum (object number) and cid (class identifier). Figure 6 shows the address format for OMU. The mapper part of OMU maps the 32-bit virtual address generated by FUs or processor to 15-bit physical address.

The main function of the packet network is invoking functional units as well as sending/receiving data and parameters to/from functional units. Any FU (hardware-implemented method) can be attached to this network. The packet network width is 61 bits. Data and parameters are transferred in the form of packets. The packet format is shown in Fig. 7. The packet-type field can have two possible values: 0 for METHOD_CALL and 1 for METHOD_DONE. When an FU wants to call another FU or a software method, it makes a METHOD_CALL packet-type. After an FU or software method is done, it sends a METHOD_DONE packet to the caller.

To invoke the packet-based method-dispatching mechanism in hardware- and software-methods, the ODYSSEY system-level synthesizer converts the virtual method calls in the input C++ program to the special routines of
A Framework for Object-Oriented Embedded System Development

<table>
<thead>
<tr>
<th>Class Identifier</th>
<th>Object Number</th>
<th>Data index</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>4 bits</td>
<td>24 bits</td>
</tr>
</tbody>
</table>

Fig. 6. The OMU address format.

<table>
<thead>
<tr>
<th>Packet type</th>
<th>Class identifier</th>
<th>Method identifier</th>
<th>Object number</th>
<th>Sender Class Identifier</th>
<th>Sender Method Identifier</th>
<th>Sender Object number</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>4 bits</td>
<td>6 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>6 bits</td>
<td>4 bits</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Fig. 7. The control packet format.

VMC\textsubscript{BY}\_HW(oid, mid) and VMC\textsubscript{BY}\_SW(oid, mid), respectively (see Fig. 1). Both functions take two parameters: the first parameter is the oid (object identifier) of the called object and the second one, mid, is the identifier of the called method. For methods that have one or more arguments, two other routines (namely, PARAMETERIZED\_VMC\textsubscript{BY}\_HW(oid, mid, params, params len) and PARAMETERIZED\_VMC\textsubscript{BY}\_SW(oid, mid, params, params len) are used instead. The params argument of these macros contains the method arguments whose length is given by the params\_len argument. Two routines are implemented to handle accesses to the objects data in our system: \texttt{OBJECT\_ATTR\_WRITE} that contacts the OMU to write a value to an attribute of an object and \texttt{OBJECT\_ATTR\_READ} that sends read request to the OMU.

6. Case Studies\textsuperscript{d}

To investigate the approach in practice, we present some case studies. These case studies consist of the design of four real-life embedded systems: JPEG and Motion JPEG encoder, JPEG and Motion JPEG decoder, and two small ones: Factorial and Determinant.

In the domain of image compression/decompression applications, the JPEG algorithm is the base for many compression/decompression algorithms, including Motion JPEG moving picture. The JPEG and Motion JPEG encoding/decoding steps are shown in Fig. 8.\textsuperscript{35} As Fig. 8 shows, if the “JPEG Data Stream Generator” (“JPEG Data Stream Parser”) is added to the JPEG encoder (decoder), Motion JPEG encoder (decoder) is implemented. As case studies, we implemented JPEG

\textsuperscript{d}All results of this section are calculated on a personal computer performing in 3 GHz and with 1 gigabyte memory.
encoder (decoder) object-oriented ASIPs and then extended the generated OO-ASIPs, using software routines to implement the Motion JPEG encoding (decoding) algorithms.

We first developed an OO program in C++ for JPEG compression/decompression. It took less than two man-months to develop and debug JPEG encoder/decoder object-oriented model from scratch. The JPEG encoder/decoder validity was verified by encoding/decoding several JPEG files produced by various JPEG-(de)compression programs as well as samples available from Independent JPEG Group and JPEG official site. In the second step, we synthesized JPEG object-oriented model by our tool and developed a JPEG Encoder/Decoder OO-ASIP. To implement Motion JPEG encoder/decoder, we extended the available JPEG object-oriented library (see Fig. 9). Development of Motion JPEG object-oriented model from JPEG library took less than 3 days due to reusing the already implemented JPEG object-oriented model. This is a key feature of our OO approach in modeling, as well as the programmable platform for implementation, which supports the reuse of the already designed and implemented blocks. In
Determinant case study, we implemented a class library that includes a hardware method calculating the determinant of a $3 \times 3$ matrix. We built its OO-ASIP and then extended this application to implement the OO-ASIP calculating determinant of a $4 \times 4$ matrix. In another simple case study, Factorial, we developed an OO-ASIP to implement multiplication, and then extended this OO-ASIP to implement the factorial. Table 3 includes case studies and number of their hardware and software methods. The first subcolumn in each column includes class library, and the second one is the extending class. OO-ASIPs are extendable merely by software methods; so as the table shows, extended classes have only more software methods.

Table 4 summarizes the synthesis results of our object-oriented applications. The synthesis is composed of two steps. The ODYSSEY synthesizer step is the system-level synthesis operations, which takes only a few seconds. The downstream synthesis step uses behavioral synthesis tools, FPGA P&R tools, and compilers to generate gate-level hardware and binary software (see Fig. 2). ODYSSEY synthesizer synthesizes the overall system; so the extended classes consume more time to synthesize than the original classes, but in Downstream synthesis it is enough to

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># of Software methods</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>11</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td># of Hardware methods</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
add software methods to the already developed OO-ASIP. Since OO-ASIP extension only requires to update a software, downstream synthesis takes only a few minutes. To obtain complete system development time, the design-time of the OO program should also be considered.

Limitations in the memory capacity of the MicroBlaze processor and its corresponding compilation tools, however, made us reduce the size of the original video frames to $32 \times 32$ pixels to enable compilation of the code for MicroBlaze and download and run it on our Virtex4 development board. We used a Xilinx ML-410 board connected to a personal computer to configure the FPGA, and also for sending/receiving data as well as for single-stepping and debugging the software being executed on the MicroBlaze processor implemented in the FPGA logic blocks.

Table 5 shows implementation results of the applications on the Virtex-4 XC4VLX60 FPGA. Device resources usage mentioned in logic data row does not include the processor logic usage (1330 slices). Hence, to obtain complete logic resource usage this value should be added to the mentioned values in the row. The Memory data row consists of two parts: the first line includes memory used for software, and the second line is the memory used by the objects’ data. Note that the higher resource usage of the hardware–software implementation is mainly due to the automatic synthesis of the hardware components (using behavioral synthesis...
### Table 6. Simulation and execution results.

<table>
<thead>
<tr>
<th>Application</th>
<th>Simulation time</th>
<th>Execution time (#cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simulation</td>
<td>Execution</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>program</td>
</tr>
<tr>
<td></td>
<td>OO Full-software</td>
<td>model</td>
</tr>
<tr>
<td></td>
<td>µs</td>
<td>µs</td>
</tr>
<tr>
<td>Base App. Multiply</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>Extending App. Factorial</td>
<td>45</td>
<td>80</td>
</tr>
<tr>
<td>Base App. Deter. 3×3</td>
<td>25</td>
<td>37</td>
</tr>
<tr>
<td>Extending App. Deter. 4×4</td>
<td>90</td>
<td>176</td>
</tr>
<tr>
<td>Base App. JPEG decoder</td>
<td>925</td>
<td>98152</td>
</tr>
<tr>
<td>Extending App. Motion JPEG decoder</td>
<td>2750</td>
<td>289300</td>
</tr>
<tr>
<td>Base App. JPEG encoder</td>
<td>990</td>
<td>37962</td>
</tr>
<tr>
<td>Extending App. Motion JPEG encoder</td>
<td>2830</td>
<td>109780</td>
</tr>
</tbody>
</table>
tools) from C++ software routines. This is indeed the known inherent problem with behavioral synthesis in spite of its merits in fully-automatic top-down implementation flow.

Table 6 summarizes the simulation and execution results of our object-oriented applications. The simulation time column reports times for the four levels of simulation that are provided in our design flow: simulating the input OO program, simulating it after hardware–software partitioning but before elaborating each partition, simulating the ISS model that co-simulates the hardware with ISS model of the processor, and finally detailed gate-level simulation. The Execution Time column gives the number of clock cycles required to simulate the applications. First part of Execution Time (OO Full-software program) includes the number of clock cycles elapsed in simulating object-oriented full-software model of applications on MicroBlaze processor.

Table 7 shows the implementation results of the ODYSSEY-introduced system functions. These functions are the OO concepts that are implemented in our OO-ASIP architecture. The second column of the table shows a number of clock cycles elapsed for calling a virtual method in a full software implementation. The third column contains the number of clock cycles needed to execute equivalent tasks in an OO-ASIP. Finally, the last column shows the memory usage of the functions.

Table 7 shows the implementation results of the ODYSSEY-introduced system functions. These functions are the OO concepts that are implemented in our OO-ASIP architecture. The second column of the table shows a number of clock cycles elapsed for calling a virtual method in a full software implementation. The third column contains the number of clock cycles needed to execute equivalent tasks in an OO-ASIP. Finally, the last column shows the memory usage of the functions.

Note that while the above area and time figures show that our OO-ASIP-based implementation is faster than the full-software one, this speedup is not its sole advantage. A major advantage of our ODYSSEY methodology and its implementation as an OO-ASIP is the extensibility through software that is provided for future extending applications. We showed such extensions in the case of the JPEG encoder and decoder OO-ASIPs being extended to the Motion JPEG encoder and decoder ones in this paper. Detailed implementation results of the case study presented in this paper yields to the conclusion that the ODYSSEY technique for “hardware evolution by software” does not impose unacceptable overheads, while it provides the flexibility and design speed offered by the software.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Delay full software (Clock cycles)</th>
<th>Delay with OMU (Clock cycles)</th>
<th>Memory usage (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual method call by software</td>
<td>26</td>
<td>75</td>
<td>$156 \times 32$</td>
</tr>
<tr>
<td>Parametrized virtual method call by software</td>
<td>29</td>
<td>80</td>
<td>$158 \times 32$</td>
</tr>
<tr>
<td>Virtual method call by hardware</td>
<td>N.A.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Parametrized virtual method call by hardware</td>
<td>N.A.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Object attribute read</td>
<td>6</td>
<td>31</td>
<td>$13 \times 32$</td>
</tr>
<tr>
<td>Object attribute write</td>
<td>5</td>
<td>36</td>
<td>$13 \times 32$</td>
</tr>
</tbody>
</table>
7. Summary and Conclusions

The main trust of this paper is to describe and discuss the implementation details and advantages of an EDA toolset that automates design and development of embedded systems in the ODYSSEY methodology. The tool promotes a smooth transition from high-level object-oriented specification to the final embedded system. This tool automatically generates hardware, software, and their interface from a given class library and application. Generated systems in intermediate levels can be simulated and verified. The final system that is an HDL synthesizable model can also be emulated by Virtex4 FPGA family.

Empirical results of real-life case studies show that although performance is not the main focus in the ODYSSEY design methodology, its implementation approach does not result in unacceptable performance overheads while its ASIP approach decreases design time due to programmability, and its OO basis brings OO advantages, such as flexibility and extensibility of design, to the hardware world. The expected lower performance compared to full-hardware implementations is compensated by the increase in the flexibility and effective reduction in the time-to-market. The case studies also showed that further improvement of the performance and area figures helps in strengthening the advantages of the methodology and the OO-ASIP architecture specially if implemented as a custom chip. The tool’s capability of incorporating predesigned IP cores instead of top-down behavioral synthesis for some functional units, gives an appropriate possibility to improve the embedded system performance. It is, however, noteworthy that the fully top-down procedure employed in case studies and the behavioral synthesis in particular, can still be cost-effective as long as the target platform is an FPGA and the given FPGA has sufficient capacity. With the current trend toward programmable platforms in very deep submicron technologies$^{33}$ and availability of multi-million gate FPGAs, this is more likely to be the case as compared to the presented case studies.

References

3. www.xilinx.com/products
4. www.celoxica.com/products
5. www.mentor.com/products
6. www.cadence.com
7. www.synopsys.com
11. R. K. Gupta, C. N. Coelho and G. Micheli, Synthesis and simulation of digital sys-

tems containing interacting hardware and software components, Proc. IEEE Design

12. A. Österling, Th. Benner, R. Ernst, D. Herrmann, Th. Scholz and W. Ye, Hard-


e-ware/Software Co-Design: Principles and Practice (Kluwer Academic Publishers,

1997).


14. J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen and A. Haxthausen, LYCOS:


The Lyngby co-synthesis system, J. Design Automation Embedded Syst. 2 (1997).


Lavagno, C. Passerone, K. Suzuki and A. Sangiovanni-Vincentelli, Hardware-Software

Co-Design of Embedded Systems — The POLIS Approach (Kluwer Academic Pub-

lishers, 1997).
16. I. Bolsens, K. van Rompaey and H. de Man, User requirements for designing complex


17. P. Chou, R. B. Ortega and G. Borriello, The chinook hardware/software co-synthesis


18. B. Dave, G. Lakshminarayana and N. K. Jha, COSYN: Hardware-software co-


19. S. Prakash and A. Parker, SOS: Synthesis of application-specific heterogeneous mul-


tiprocessor systems, J. Parallel Distributed Comp. 16 (1992) 338–351.
20. C. Valderrama et al., Hardware and Software Co-Design: Principles and Practice


21. J. S. Young et al., Design and specification of embedded systems in Java using suc-


23. Ptolemy Project on Heterogeneous Modeling and Design, Online Home Page, Feb


2007, http://ptolemy.eecs.berkeley.edu
24. C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn and W. Rosenstiel, Object-


oriented modeling and synthesis of systemC specifications, Proc. Asia South Pacific

25. E. Grimpe, B. Timmermann, T. Fandrey, R. Binisch and F. Oppenheimer, SystemC


object-oriented extensions and synthesis features, Forum on Design and Specification

Languages (2002).


fusion.com
27. F. Cheng and H. Wu, Design and implementation of software objects in hardware,


28. SOCAD, A CAD tool for SOC Design, 4C applied technologies lab. of CSE Dept.


Tatung University, Taiwan, http://4c.cse.ttu.edu.tw/snipsnap/space/Socad


public.itrs.net/
31. K. Keutzer, S. Malik and A. R. Newton, FromASIC to ASIP: Thelnest design dis-


32. L. Benini and G. L. De Micheli, Networks on chips: A new soc paradigm, J. IEEE


33. N. Zeinolabedini, Development of a co-simulation environment in a system level design methodology, MSc thesis, Sharif University of Technology, Tehran, Iran (2006).
34. S. Hashemi, Utilizing IP cores at register transfer level for implementing hardware methods in OO-ASIP, MSc thesis, Sharif University of Technology, Tehran, Iran (2006).