Combining Processor Virtualization and Component-Based Engineering in C for Many-Core Heterogeneous Embedded MPSoCs

Erven Rohou, Andrea Carlo Ornstein, Ali Erdem Özcan, Marco Cornero

Combining Processor Virtualization and Component-Based Engineering in C for Many-Core Heterogeneous Embedded MPSoCs

Erven Rohou*, Andrea C. Ornstein†, Ali Erdem Özcan†, and Marco Cornero‡

*INRIA, †STMicroelectronics, ‡ST-Ericsson

Abstract

The design of embedded systems is driven by strong constraints in terms of performance, silicon area and power consumption, as well as pressure on the cost and time-to-market. This has three consequences: 1) many-core systems are becoming mainstream, but there is still no satisfactory approach for distributing software applications on these platforms; 2) these systems integrate heterogeneous processors for efficiency reasons, thus programming them requires complex compilation environments; 3) hardware resources are precious and low-level languages are still a must to exploit them fully. These factors negatively impact the programmability of many-core platforms and limit our ability to address the challenges of the next decade.

This paper devises a new programming approach leveraging processor virtualization and component-based software engineering paradigms to address these issues all together. We present a programming model based on C for developing fine grain component-based applications and a toolset that compiles them into a processor-independent bytecode representation that can be deployed on heterogeneous MPSoCs. We also discuss the effectiveness of this approach and present future directions that will have a key role in addressing the above challenges.

I. Introduction

Compared to the general purpose computing world, embedded consumer systems are characterized by particularly stringent efficiency requirements in terms of performance, power and area. SoC (System-on-Chip) vendors used to design application specific hardware components (IPs) controlled by a *host* processor for implementing complex operations. This process is no longer valid because of today's technological and market constraints. Non-recurring engineering costs of silicon manufacturing are becoming so high that the same IP must be reused in many products. Short time-to-market also plays in favor of reuse. In other words, hardware needs to be more programmable. In this context, the design of software-based solution substituting the hardware IPs becomes one of the most critical topics of the overall production process of SoCs.

Problem Statement Unfortunately, designing software systems for those constrained platforms is a complex issue. The reasons can be categorized as follows.

1) *Large scale multi-processing is a must.* For almost 30 years, general purpose processors as well as embedded systems followed Moore's law. Performance used to "automatically" double every 18-24 months. However, since 2002, diminishing performance return, as well as increasing power consumption, and the approaching "temperature wall" made the microprocessor industry follow a new path for performance: the multicore approach. While it is true that embedded system have always been heterogeneous multicores, several studies take parallelism to unprecedented levels and forecast thousands of cores on a chip by the end of the next decade [1], [2]. Programming distributed applications has always been difficult, even in general purpose computing, because of the lack of convenient programming abstractions and tools [3]. The situation is worse for SoC platforms, with minimum runtime support, and no alternative to low-level programming languages. This has always constituted a big barrier to the productivity. Given the new order of magnitude of cores, the exploitation of the available hardware is a major challenge of the coming decade.

2) *Heterogeneity is unavoidable.* Although the homogeneization of hardware platforms aims at reducing production costs, modern embedded platforms continue to integrate heterogeneous computing nodes (e.g. DSP, VLIW, etc.) for several reasons. First, suitable hardware support (i.e. instruction sets, data representation, etc.) is key to satisfying the performance requirements of different kinds of applications executed on a single chip. Second, these computing nodes are always evolving to enjoy the best

technological solutions in the market. Beside evolving architectures, heterogeneity is a source of difficulties since it requires the software development kits (compilers, debuggers, profilers, etc.) to be ported and maintained, and the developers to be trained on them. Tools for different cores are likely to come from different vendors and to use different technologies, yielding to integration problems. The source code is likely to be written with conditional compilation directives (`#ifdef`) in order to adapt to each compiler and to best exploit each architecture. While they are of no theoretical nature, these issues are a significant burden (and cost) to software companies.

3) *Physical resources are precious.* Embedding software in consumer systems has always been a challenge for satisfying the performance requirements on top of limited constrained physical resources. Even though an unprecedented number of cores is expected soon, silicon area (be it CPU or memory) directly translates into cost. Each core must be as efficient as possible. Henceforth, software developers are still not free to enjoy high-level programming languages and run-time environments such as garbage collectors, exception handlers, and rich libraries. This results in many applications being coded from scratch and reduces the productivity while increasing the maintenance, support and evolution costs.

These issues motivate many language-oriented research projects [4], [5]. However, some pragmatic concerns (legacy, efficiency, toolset availability, etc.) make industrial solutions evolve with small steps. In particular, since many multimedia processors do not support higher level languages such as C++ or Java, the C language remains the *de facto* standard, although it does not sufficiently deal with heterogeneity and multi-processing issues. Indeed, the C language implies the use of different development kits (compiler, debugger, etc.), potentially coming from different vendors, for each processor. Beside the overhead related to their installation and maintenance, these tools may behave differently (command-line flags, error messages, etc.), and may imply the source code to be specialized for their own header files, intrinsics and library functions[1]. In addition, the lack of specific abstractions for application distribution makes plain C programs very difficult to deploy efficiently on multiple processors.

Contribution In the past, some research and industrial proposals partially addressed these issues. On the one hand, processor virtualization provided a way of dealing with heterogeneous target platforms. On the other hand, component-based software engineering approach improved the software modularity and contributed to the development of distributed systems.

The main contribution of this paper is the design and implementation of a toolset combining these two paradigms, for the C language, in order to start addressing the above issues all together. Using this toolset, *legacy code* written in C can be encapsulated into well-defined components programming model with limited re-engineering effort. These components can then be composed using an Architecture Description Language (ADL) that describes the software architecture of the system, and compiled into a target independent bytecode representation. Using this toolset, programmers can develop reusable binary component libraries that can be used by system architects for composing applications to be deployed on heterogeneous multiprocessor SoC (MPSoC). *Newly developed* components, on the other hand, can be envisioned either in C or in any higher-level language that can be compiled to the bytecode representation.

In addition, the combination of component-based design with such a target independent bytecode representation opens various perspectives that leverage the performance of embedded systems. First, the design flexibility can be increased by mapping components on heterogeneous processors at *deployment* time. In particular, this lets programmers deal with memory hierarchy as an orthogonal issue. Second, important memory and performance optimizations can be obtained by on-board generation of interface-specific communication bridges between remote components.

Outline This paper is organized as follows. Section II overviews the paradigms we propose to combine, and it presents the state-of-the-art on these domains. Section III presents our proposal. Section IV discusses its effectiveness and presents some ideas that we will investigate in future work. Finally, we conclude in Section V.

II. Related Work

This paper proposes a new methodology and toolset for programming heterogeneous MPSoCs in the C language. We build upon previous works on processor virtualization and component based software engineering. This section reviews previous work related to both paradigms.

A. Processor Virtualization

Processor virtualization first appeared for deploying programs on computers connected through the Internet, and became a well established technique for dealing with the processor heterogeneity. While the traditional compilation flow consists in compiling program sources into binary objects that can be natively executed on a given processor, processor virtualization splits that flow in two parts. The first part — the *front-end* — compiles the program source code into a processor-independent bytecode representation.

[1]Target-specific source-code specialization makes the code difficult to maintain and hinders debugging on the workstation.

The second part — the *back-end* — provides an execution platform to run this bytecode on a given processor. The *back-end* may either be a virtual machine interpreting the bytecode or a dynamic compiler translating the processor-independent bytecode to native binary at load time or run time in order to improve the execution performance. This split of the compilation flow has many benefits for dealing with the heterogeneity issue. First, developers can use the *same* development kit for compiling their programs on their workstation and debugging them on the platform. Second, the same bytecode can be loaded on a heterogeneous MPSoC, and the decision of the processor on which it will execute can be postponed until run time. Finally, split-compilation lets the back-end apply aggressive optimizations thanks to additional information computed offline by the front-end [6].

The Java framework defines a bytecode-based virtual machine and a standard library for the Java language. The lightweight version of Java, namely Java Micro Edition, has been widely accepted in heterogeneous embedded systems in order to provide complementary capabilities, like games for cell phones or TV guides for set-top-boxes. Its use remains constrained to the host processor for the non-critical part of the application. Components models are supported (see Section II-B), but the C language cannot be compiled to the Java bytecode.

LLVM [7] is a compiler framework that defines a low-level code representation appropriate for program analysis and transformation. This representation is intended to be processor- and input language-independent, and well suited for optimized code generation. Many languages, including C, can be compiled to LLVM. However, the representation is at a lower level than CLI. In addition, it has not been standardized, and, as such, is subject to changes.

The Common Language Infrastructure (CLI) [8] is an international standard that defines a rich virtualization environment for the execution of applications written in multiple languages. Beside the .NET Framework and the .NET Micro Framework provided by Microsoft, there exist several open-source programming environments based on CLI. Component models are supported. We have previously contributed GCC4CLI [9], a C compiler that generates efficient CLI code for embedded systems. We have shown that CLI is an appropriate format for the deployment of embedded software, in terms of both code size [10] and performance [11]. We also demonstrated several flexible compilation flows from C to CLI [12].

B. Component-based Software Engineering

Although the foundations of composing the software systems by assembling components appeared very early [13], component-based programming has been widely accepted as a new programming paradigm in the last decade for succeeding the object-oriented programming [14]. In a nutshell, component-based programming is about structuring the software modules as independent components that fulfill well-defined specifications in terms of client (required services) and server (provided services) interfaces. The strong encapsulation of data and behavior and the capture of the software architecture in terms of components, interfaces and their interconnections makes this approach suitable for distributing complex applications on multiple processors. Furthermore, these features allow the use of many appropriate design tools supporting the assembly, the verification and the distributed deployment of components using a description of the software architecture (ADL).

Many component models have been used in general purpose computing during the last decade for improving program modularity and managing software distribution. The most adopted ones include the COM family (COM, COM+, DCOM) from Microsoft, the CORBA Component Model (CCM) from OMG, the Enterprise Java Beans (EJB) from Sun Microsystems and the Open Services Gateway Initiative from OSGI Alliance. These component models are in general tailored for powerful workstation environments and most of the services that they implement (e.g consistency, security, failure recovery, etc.) fit neither the requirements nor the computational budget of MPSoCs.

Component-based programming has also appeared in embedded platforms in recent years. Real Time Software Components [15] (RTSC) from Texas Instruments provides an ADL toolset based on JavaScript and a packaging format for building modular system software from component libraries. OpenMAX [16] from Khronos provides a component-based middleware for easing the integration of audio/video codecs for building complex multimedia applications. Both models are based on the C language, and aim at improving the software reuse. Nevertheless, they are designed for single processor systems.

There exist other component technologies especially designed for MPSoCs. DSOC [17] is a light-weight implementation of CORBA. It provides a toolset that generates middleware components in hardware for accelerating inter-processor communications over a given network on chip. Cecilia [18] is a C-based implementation of the Fractal Component Model [19] and provides a deployment environment for distributing multimedia components on heterogeneous MPSoC platforms. In addition, Cecilia provides an extensible ADL toolset [20] that allows the integration of new code generation features by third party developers. Khronos recently proposed a new standard, called Open Computing Language (OpenCL) [21], for programming parallel graphics applications. OpenCL provides extensions to the C language for mapping the program data to a hierarchical memory architecture and for using a standard

interface to vectorial instructions. Therefore, it requires either the Clang OpenCL compiler combined with the special runtime support for its bytecode on the target processors, or the target C compiler to be modified for supporting these extensions.

C. Combining Virtualization and Component-based Engineering in C

While several approaches have been proposed both for processor virtualization and for component-based software engineering, to the best of our knowledge there is no prior art considering their combination with the C language.

We believe that this combination provides a promising groundwork for addressing the code generation challenges of the decade to come. That is, (i) component-based software engineering provides a suitable way of structuring software systems to be deployed on distributed systems, such as MPSoCs, (ii) virtualization helps dealing with the heterogeneity of target platforms without imposing the burden of binary compatibility on hardware designers, and (iii) the C language, which is the *de facto* standard for programming embedded systems, allows writing low-level code without any expensive run-time dependencies.

III. Our Proposal

Our proposal, namely Virtual Components for SoC (VC4SoC), consists in a programming toolset that enables target-independent component-based programming in C. The generated libraries are for deployment on heterogeneous MPSoCs. We first review the VC4SoC compilation flow. Then, we present its key features including the binary layout of components, the programming model used for implementing them in C, and the specific linker that maps the component implementations to the binary layout.

A. Compilation Flow

As depicted in Fig.1, the development flow with VC4SoC consists of two parts: *front-end* and *back-end*. The *front-end* compiles source components into binary component libraries. It runs on a workstation. The *back-end* is about the composition of applications from binary component libraries. It runs either on a workstation for static mapping of applications on a target platform, or on board to enjoy dynamic mapping at deployment time.

The *front-end* development starts with the specification of components (0). Two specific languages are involved to improve the robustness of these specifications compared to hand-written C header files. First, a strongly-typed *Interface Definition Language* (IDL) specifies the methods that can be used for the interaction between components.

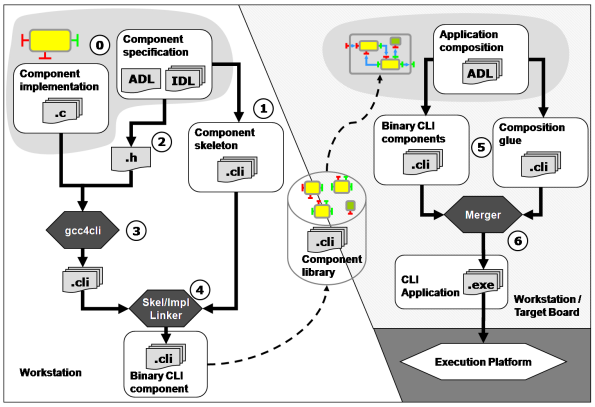

Fig. 1. Overview of the compilation flow

The IDL allows many static verifications to be done and part of the interaction code to be generated. Second, an *Architecture Description Language* (ADL) specifies the set of interfaces that are provided and required by components, as well as their interconnections. The expressiveness of the ADL eases the description of complex composition schemes. Static verifications can be performed and the glue code to assemble the components can be generated.

These specification files are used for generating two separate files. The *skeleton* of the component is generated in CLI (1) in respect to its architecture specification. This *skeleton* defines the binary layout of a CLI object representing the component without the implementation of its interfaces. The implementation is provided by the programmer in C using a lightweight programming model based on standard pre-processing macros. These macros are used to access component's members such as client interfaces or private data fields. This implementation is completed with a generated header file containing the values of these macros, according to the architecture specifications (2). The *implementation* source files are then compiled to CLI (3) using a C-to-CLI compiler [9], [12]. Finally, a specific linker, described in Section III-B3 merges the *skeleton* of the component with its *implementation* (4).

The role of the *back-end* is to compose applications from a binary component library output by the *front-end*. The input driving this process is an ADL description specifying the top-level architecture of the application to be composed. Based on this description and the architecture specifications reflected by the binary components in CLI, the *back-end* generates the glue code that is required for assembling the application components and mapping them to the target execution platform (5). A typical example for glue code generated within the context of MPSoCs may be the inter-processor communication channels implementing the interactions between remote components. The glue code is directly generated in CLI on the target platform

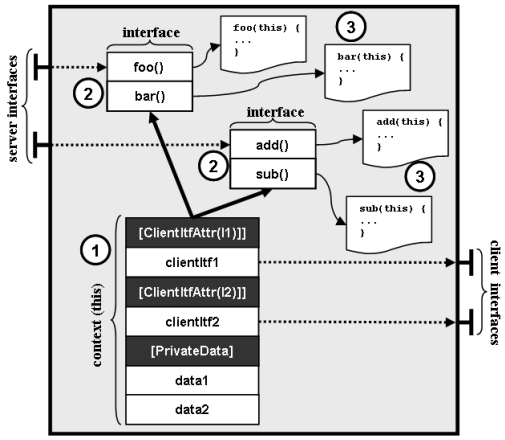

Fig. 2. Binary structure of a component in CLI

using lightweight bytecode manipulation tools such as Cecil [22]. Finally, a *merger* tool can be used for gathering the components that go on to the same processor in order to optimize their deployment on the target platform (6).

B. Component model

VC4SoC is based on the Cecilia Component Framework[2] [18], a lightweight implementation of the Fractal Component Model [24], [25] dedicated to the development of embedded applications and systems. This section first presents the binary layout of VC4SoC components. Then it describes the programming model and the linker tool that is designed for encapsulating standard C programs into these components.

1) Binary layout: VC4SoC provides a cost-effective implementation of components based on unmanaged CLI structures. As depicted in Fig.2, a component at run time consists of a data structure in addition to the implementation of its methods. This is similar to the native layout of a C++ object. The main part of this data structure, namely *component context*, contains fields for implementing the set of *client interfaces* of the component as well as its private data members. In addition, the *component context* inherits *interface structures* for each of its *server interfaces*. These *interface structures* contain pointers giving access to the implementation of the methods implemented by the component.

In addition to the above functional fields, the *component context* structure is annotated with CLI metadata attributes giving supplementary information about the component's architecture (e.g. interface names, component name, etc.) This information is required by the *back-end* of the VC4SoC infrastructure for composing applications from binary component libraries. This metadata can be removed at run time to save memory or be kept to enjoy reflective programming.

Note that only the *component context* needs to be duplicated when a new instance of a component is created. Hence, the performance overhead of VC4SoC components is comparable to C++ objects, and is expected to be even less since they require only unmanaged CLI structures.

2) Programming model: VC4SoC defines a programming model for encapsulating standard C programs into the above component model. This programming model consists of few pre-processing macros that define handles for linking the *implementation* code written by the programmer with the *skeleton* code generated by the ADL compiler.

Fig.3-b illustrates the implementation of a simple component printing a message on a printer interface. `PRIVATE_DATA` and `DATA` macros used for declaring and accessing the instance data structure of the component, respectively. The `METHOD` macro is used for declaring the implementation of a method provided by the component. Finally, the `CLIENT` macro is used for referencing a client to be invoked.

The implementation code depicted in Fig.3-b needs to be completed with the definition of the above macros to become a correct C program. For that purpose, the *front-end* of the VC4SoC compiler generates a header file from the ADL of the component. As depicted in Fig.3-a, this header file starts with the definition of `PRIVATE_DATA` macro which defines the *component context* structure and declares an instance of the latter (`_this` variable). `DATA` and `CLIENT` macros define accesses to this `context` variable. The `METHOD` macro mangles the name of the implemented method with the name interface and the component to which it belongs to avoid name clashes.

Careful readers may have noticed that the type definitions in the above header file do not match the binary layout presented in Fig.2. In particular, the *component context* does not inherit from the server interface structures and the global definition of `_this` variable makes the component a singleton instance. Indeed, these are placeholders (*fake definitions*) intended to be substituted at link-time time by the component skeleton, generated in CLI. These placeholders are used for two reasons: first, they complete the implementation code and make it legal C code, that can be compiled with a standard C compiler; and second, they mark some fields with specific compiler attributes (e.g. *ClientInterface*) so that these fields can be easily located by the *skeleton/implementation* linker.

The benefits of generating the component skeletons in CLI rather than in C are twofold. First, CLI provides native support for modeling component interfaces and annotating data structures with additional information. This way,

[2]Cecilia was formerly called Think [23].

```
#define PRIVATE_DATA \
  MyComp_data_t ; \
  typedef struct { \
     struct Printer p; \
     MyComp_data_t data; \
  } MyComp_context; \
  MyComp_context _this \
    __attribute__ ((cli))
#define METHOD(itf, meth) \
  __attribute__ ((cli)) MyComp_##itf##_##meth
#define CLIENT _this
#define DATA _this.data
```

(a)

```
typedef struct {
  char *message;
} PRIVATE_DATA;
int METHOD(m,main) (){
  DATA.message = "hello world";
  CLIENT.p.print(DATA.message);
}
```

(b)

```
public interface MainInterface {
    int main() ;
}

public interface Printer{
  void print(string msg);
} ;

[PrivateDataStructure]
struct MyComp_data_t  {} ;

public unsafe struct MyComp_context
  : MainInterface{
  [ClientInterface("p")]
  Printer p ;

  [PrivateData]
  MyComp_data_t data;

  public int main(){return 0;}
};
```

(c)

Fig. 3. Excerpt of hand-written and generated files. (a) Generated header file, (b) hand-written component implementation, (c) generated component skeleton.

the architecture information is natively embedded in the binary, without the need for any adhoc extension. Second, CLI enables the inheritance of interface structures within data structures for modeling the interfaces implemented by an object. As depicted in Fig.3-c[3], the component skeleton uses these features for defining the *component context* structure matching exactly the binary layout presented in Section III-B1. On the other hand, empty bodies are generated for component's private data members and method implementations. The latters are intended to be filled with the information coming from component's *implementation* by the linker described in the following section.

3) Skeleton/implementation linker: As described previously, during the first part of the compilation flow, the hand-written *implementation* in C is compiled to CLI, and the *skeleton* is directly generated in CLI. They still need to be linked together in order to form a binary component. This is done by a specific linker which completes the binary skeleton file with the information coming from the *implementation* file.

The steps of this link process are as follows.

1) The definition of the private data members of the component is copied from the *implementation* file into the skeleton. This operation is straightforward since all data members are gathered in a data structure definition whose name conforms to a convention translated by the `PRIVATE_DATA` macro.
2) The method bodies are copied from the *implementation* file into the skeleton. This operation is straightforward as well since the methods to be copied are annotated using specific attributes (e.g. `cli` attribute in Fig. 3-a).
3) In each method body, references to the fake `_this` variable are replaced by accesses to private data members defined in the first step. This way, the singleton content definition that uses a global context variable `_this` (Fig. 3-b) is turned into standard CLI containing method implementations from which multiple instances can be created at run time.
4) Finally, a new argument is added to each method call found in method bodies in order to specify the target of the invocation. The convention used for translating the `CLIENT` macro helps the recognition of whether the call is directed to the component itself or to a one of its client interfaces.

Note that, the *skeleton/implementation* linker is processor-independent since it only manipulates the CLI representation. Moreover, its implementation required limited effort thanks to the use of an existing CLI manipulation tool, namely Cecil. Such a CLI manipulation tool can also be used in the *back-end* of the compilation flow in order to implement optimizations such the removal of CLI attributes and the merge of multiple components into one.

IV. Discussion

A. Analysis

We have implemented the VC4SoC compiler as an extension of the Fractal ADL Compiler [20]. It accepts as input a top-level architecture description of the assembly of components for building an application. The output is an executable object in CLI. This object can then be executed on any CLI-compliant platform.

[3]Note that, for readability purposes, we used C# syntax to illustrate the generated skeleton. The actual skeleton is directly generated in CLI.

Exhaustive performance analysis would have to be performed on different heterogeneous MPSoC platforms for verifying that applications of interest developed using VC4SoC do not raise significant overheads in terms of execution time and memory footprint. Yet, previous results on component-based programming and processor virtualization paradigms — both well established — have separately demonstrated that they have acceptable overheads within the context of embedded systems. In particular, Özcan et al. [26] has shown that the fine-grained component-based re-engineering of an H.264 video decoder using the Fractal Component Model in C resulted in 1.5% overhead in execution time and about 7% overhead in memory footprint while easing notably the distribution of the decoder on multiple processors. The potential direct cost is similar to the penalty object-oriented languages incur. Moreover, Fassino et al. [23] has shown that modular design with component-based programming may also result in better performance by designing on-demand software systems. Efficient CLI code can be generated from the C language [10], [11], and robust and flexible compilers and toolchains are available. The performance achieved with these schemes is similar to native code [9], [12].

It is also important to note that the performance of applications developed with VC4SoC is extremely related to the performance of the target execution platform. Previous studies [27] show that, in the case of DSP or VLIW processors, advanced just-in-time compilation techniques may result in even more optimized code compared to static native compilation.

A limited performance penalty must also be contrasted with the significant gain in productivity and the additional flexibility brought by our approach.

Our proposal does not directly address memory hierarchy issues. However, we provide the means for the programmer to postpone the processor mapping decisions until deployment time, when the actual memory hierarchy is known. In this respect, we do not provide the policy, but the tools to implement it.

B. Limitations

The approach we presented in this paper has two main limitations that may impact the reuse of existing legacy. The first one is related to the use of a specific C compiler (GCC4CLI [9]) that we extended to recognize some annotations. In particular, we took advantage of the GCC attribute mechanism. Legacy code that uses language extensions or compiler features from another vendor might be a problem, even though we feel that our contribution to the simplification of the development environment makes the porting worth the effort in many cases. In any way, annotations are essentially used as markers that must be propagated unmodified in the CLI representation. Any other mechanism that achieves this goal can be considered. For example, we preferred attributes over pragmas simply because the GCC internals discourage the use of pragmas.

The second limitation is related to the use of processor-independent bytecode. Although the latter is unavoidable for portability on heterogeneous platforms, the use of inline assembly (`asm` blocks) introducing processor-specific optimizations in the C code are not supported anymore. Note that it is still possible to invoke native binary code (libraries, or hand-optimized routines) from CLI components thanks to the *pinvoke* mechanism [8]. In any case, this latter approach is superior in terms or software engineering: code readability, portability, and maintenance.

C. Perspectives

The combination of processor virtualization with component-based programming opens many perspectives for embedded applications. Two of them, which we plan to investigate as future work, are discussed hereafter.

1) On-board component assembly: Current practice of quality-of-service implementation on MPSoCs consists of switching between different combinations of statically linked application mappings according to the application scenarios executed by the end-user. On heterogeneous platforms, the number of combinations becomes very significant. Henceforth, the mapping flexibility is often limited by the memory budget.

Splitting the compilation life cycle into two phases as presented in Section III-A is key to break the above limitation. This way, applications can be assembled on board just before their deployment. Furthermore, as VC4SoC components can be deployed on any processor, any application mapping can be produced at run time from a given component library.

2) On-board communication adapter generation: Automatic generation of communication adapters from IDL descriptions is a well established technique for implementing remote interactions on multi-processor systems. Nevertheless, current tools designed for the C language require a workstation to generate and compile such components. As a result, programmers must forecast the possible application mappings on different processors and specify the interfaces for which adapters must be generated. Yet, this constitutes an obstacle to the mapping flexibility.

VC4SoC components provide two features that are key to solve this issue. First, binary components can be introspected to access their interface specifications. Second, CLI bytecode can be directly generated on board using bytecode manipulation tools such as Cecil. Using these features, communication adapters can be generated dynamically when they are needed. The combination of

such a tool with an on board component assembler, as discussed above, may result in a very flexible execution environment for building performance effective solutions based on heterogeneous multi-processor platforms.

V. Conclusion

This paper proposes a new programming methodology and toolset to improve the software development practice on many-core heterogeneous MPSoCs. Our approach has its roots in two well established paradigms, namely component-based software engineering and processor virtualization. Using this toolset, programmers can encapsulate legacy C code into software components with limited effort. The use of component-based programming enables the assembly of complex applications from reusable component libraries and the generation of glue code for mapping these components on a multi-processor platform. Thanks to the CLI processor virtualization technology, binary components that are output by this toolset can be deployed on any target processor of a heterogeneous system. Moreover, the latter provides programmers with an homogeneous development platform by offering them a single virtual target for all cores present on an MPSoC. This greatly simplifies the software engineering process and reduces the burden (thus, the cost) associated with software development for multiple targets or heterogeneous targets. We believe that this combination of paradigms provides a promising groundwork to address the code generation challenges of the future MPSoC platforms by defining a framework for the development of C-based applications for heterogeneous many-core platforms.

Our proposal also opens up several opportunities, which we will explore as future work. First, it makes it possible to postpone the deployment choices to run time, giving more flexibility to system designers. Second, communication adapters between components can be generated on board, and precisely tuned for the current mapping, instead of being statically generated for a given set of envisioned mappings.

References

[1] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[2] K. De Bosschere, W. Luk, X. Martorell, N. Navarro, M. O'Boyle, D. Pnevmatikatos, A. Ramírez, P. Sainrat, A. Seznec, P. Stenström, and O. Temam, *High-Performance Embedded Architecture and Compilation Roadmap*, ser. LNCS, 2007, vol. 4050, pp. 5–29.

[3] T. Mattson and M. Wrinn, "Parallel programming: can we please get it right this time?" in *DAC '08: Proc. 45th Annual Conference on Design Automation*, 2008, pp. 7–11.

[4] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu, "Spidle: a DSL approach to specifying streaming applications," in *Proc. 2nd international conference on Generative programming and component engineering*, 2003, pp. 1–17.

[5] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[6] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," in *Proc. 47th Design Automation Conference (DAC 2010)*, Jun. 2010.

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. International Symposium on Code Generation and Optimization*, 2004.

[8] *International Standard ISO/IEC 23271:2006 - Common Language Infrastructure (CLI), Partitions I to VI*, 2nd ed., International Organization for Standardization and International Electrotechnical Commission.

[9] R. Costa, A. Ornstein, and E. Rohou, "CLI Back-End in GCC," in *GCC Developers' Summit*, Ottawa, Canada, Jul. 2007, pp. 111–116.

[10] R. Costa and E. Rohou, "Comparing the size of .NET applications with native code," in *CODES+ISSS*, 2005, pp. 99–104.

[11] M. Cornero, R. Costa, R. Fernández Pascual, A. Ornstein, and E. Rohou, "An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems," in *International Conference on HiPEAC*, ser. LNCS, vol. 4917. Göteborg, Sweden: Springer Verlag, Jan. 2008, pp. 130–144.

[12] E. Rohou, A. C. Ornstein, and M. Cornero, "CLI-Based Compilation Flows for the C Language," in *Proc. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, Jul. 2010, pp. 162–169.

[13] M. D. McIlroy, "Mass produced software components," in *Proc. NATO Software Engineering Conference*, 1968, pp. 138–155.

[14] C. Szyperski, *Component software: beyond object-oriented programming*, 1998.

[15] "Real-Time Software Components," http://wiki.eclipse.org/DSDP/RTSC.

[16] "OpenMAX Development Layer API Specification, Version 1.0.1," The Khronos Group Inc, Tech. Rep., Jun. 2006.

[17] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonnard, B. Lavigueur, and D. Lo, "Distributed object models for multi-processor SoC's, with application to low-power multimedia wireless systems," in *DATE*, 2006, pp. 482–487.

[18] "Cecilia web site," http://fractal.objectweb.org/cecilia-site/current/.

[19] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The Fractal Composition Framework, *The ObjectWeb Consortium - Interface Specification*," http://www.objectweb.org, June, 2002.

[20] M. Leclercq, A. E. Özcan, V. Quéma, and J.-B. Stefani, "Supporting heterogeneous architecture descriptions in an extensible toolset," in *29th International Conference on Software Engineering*, May 2007.

[21] A. Munshi, "The OpenCL specification version 1.0," Khronos OpenCL Working Group, Tech. Rep., 2009.04.02.

[22] "Cecil Library," http://www.mono-project.com/Cecil.

[23] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, "THINK: A Software Framework for Component-based Operating System Kernels," in *USENIX Annual Technical Conference*, 2002.

[24] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal Component Model and its Support in Java," *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, vol. 36, pp. 1257–1284, 2006.

[25] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The Fractal Component Model, v2," 2003.

[26] A. E. Özcan, O. Layaida, and J.-B. Stefani, "A Component-based Approach for MPSoC SW Design: Experience with OS Customization for H.264 Decoding," in *Proc. 3rd Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2005*. IEEE Computer Society, 2005, pp. 95–100.

[27] B. Dupont de Dinechin, "Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors," in *The 14th International Euro-Par Conference on Parallel and Distributed Computing*, Las Palmas de Gran Canaria, Spain, Aug. 2008.