

## Combining Processor Virtualization and Component-Based Engineering in C for Heterogeneous Many-Core Platforms

Erven Rohou<sup>\*</sup>, Andrea Carlo Ornstein<sup>\*\*</sup>, Ali Erdem Özcan<sup>\*\*</sup>, Marco Cornero<sup>\*\*\*</sup>

**Abstract:** Embedded system design is driven by strong efficiency constraints in terms of performance, silicon area and power consumption, as well as pressure on the cost and time-to-market. As of today, this forms three tough problems: 1) many-core systems are becoming mainstream, however there is still no decent approach for distributing software applications on those platforms; 2) these systems still 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 subtly. These factors have negative impact on 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 technologies to address these issues all together. It presents 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 platforms. We also discuss the effectiveness of this approach and present some ideas that might have a key role in addressing the above challenges.

**Key-words:** virtualization, bytecode, just-in-time compilation, multicore, many-core, software engineering, software components

---

### *Combinaison des techniques de virtualisation et de programmation à base de composants en C pour les plates-formes multicœur hétérogènes*

**Résumé :** *La conception de systèmes embarqués est soumise à de fortes contraintes en termes de performance, surface de silicium, consommation électrique, ainsi que la nécessité de maintenir les coûts bas et de réduire le temps de mise sur le marché. Aujourd'hui, ces contraintes se concrétisent sous la forme de trois problèmes essentiels : 1) les plates-formes à base de processeurs multicœur deviennent courantes, bien qu'il n'existe toujours aucune approche satisfaisante pour y distribuer les applications logicielles; 2) ces plates-formes continuent d'intégrer des processeurs hétérogènes pour des raisons d'efficacité, ce qui rend leurs environnements de programmation encore plus compliqués; 3) enfin, les ressources matérielles étant extrêmement précieuses en raison des contraintes sus-citées, l'utilisation des langages de programmation de bas niveau s'impose pour assurer leur exploitation fine. Ces facteurs ont un impact négatif sur la programmabilité des plates-formes multicœur, et constituent un obstacle majeur menaçant notre capacité à nous attaquer aux défis de la prochaine décennie.*

*Ce rapport présente un élément de réponse pour aborder ces problèmes. Notre proposition se présente sous la forme d'une nouvelle approche de la programmation s'appuyant sur des techniques de virtualisation de processeur et de génie logiciel à base de composants. Nous présentons un modèle de programmation basé sur le langage C qui permet la construction d'applications à partir de composants à grain fin. Ce modèle de programmation est supporté par une chaîne d'outils qui compile les composants logiciels vers une représentation bytecode indépendante du processeur cible, ce qui les rend susceptibles d'être déployés sur des plates-formes multicœur hétérogènes. Nous discutons l'efficacité de notre approche et nous présentons plusieurs perspectives de recherche qui pourraient avoir des rôles clés dans la résolution des problèmes sus-cités.*

**Mots clés :** *virtualisation, bytecode, compilation à la volée, multicœur, génie logiciel, composants logiciels*

---

This work was mostly done while the authors were at STMicroelectronics.

\* Projet Alf: équipe commune INRIA et université de Rennes 1

\*\* STMicroelectronics

\*\*\* ST-Ericsson

# 1 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. In the past, SoC (System-on-Chip) vendors were designing specific hardware components (IPs) controlled by a *host* processor for implementing complex operations, and they were scaling the clock frequency for satisfying the performance requirements. Nevertheless, this process is no more 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. Shorter time-to-market constraints also plays in favor of reuse. In other words, hardware needs to be more adaptable. 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. This is mostly due to the following reasons:

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 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 independent studies take parallelism to unprecedented levels and forecast thousands of cores on a chip by the end of the next decade [1, 8, 13]. Programming distributed applications has always been an issue, even in general purpose computing, because of the lack of convenient programming abstractions and tools [25]. 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 for the productivity. Given the new order of magnitude of available cores, the exploitation of the available hardware is a major challenge of the coming decade.
2. *Heterogeneity is unavoidable.* Although the design for manufacturability efforts aim at the homogeneization of hardware platforms for reducing production costs, modern embedded platforms continue to integrate heterogeneous computing nodes (DSP, VLIW, etc.) for several reasons. First, suitable hardware support (instruction sets, data representation, etc.) is key to satisfying the performance requirements of different kinds of applications executed on a single chip. Second, the versions of these computing nodes are always evolving to enjoy the best technological solutions in the market. Beside the need for porting the applications for these evolving architectures, the heterogeneity is a big source of issue since it requires the software development kits (compilers, debuggers, profilers) to be ported and maintained, and the developers to be trained for using these tools. 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 of the many cores is expected to be as efficient as possible. Henceforth, software developers are still not free to enjoy high-level programming languages and runtime 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 such as Spidle [9], StreamIt [40], X10 [31] and Intel's Concurrent Collections [20]. However, some pragmatic concerns (legacy, efficiency, toolset availability, etc.) make industrial solutions evolve with small steps. In particular, the C language remains a *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), and may imply the source code to be specialized for their own header files, intrinsics and library functions<sup>1</sup>. 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. In particular, processor virtualization technologies 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 technologies, in order to start addressing the above issues all together. Thanks to this toolset, *legacy code* written in the C language can be encapsulated into well-defined components using a macro-based programming model with limited re-engineering effort. These components can then be composed using an Architecture Description Language (ADL) and compiled into a target independent bytecode representation. Using this toolset, programmers can develop truly 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 byte code representation opens various perspectives that leverage the performance of embedded systems. First, the design flexibility can be increased by mapping components

<sup>1</sup>Target specific source code specialization makes the code difficult to maintain and inhibits the final binaries to be debugged on the workstation.

on heterogeneous processors at *deployment* time. Second, important memory and performance optimizations can be obtained by on-board generation of interface-specific communication bridges between remote components.

**Outline** Section 2 overviews the technologies we propose to integrate, and it presents the state-of-the-art. Section 3 presents our proposal. Section 4 discusses its effectiveness and presents some future work. Finally, Section 5 concludes the paper.

## 2 Background and Related Work

This paper proposes a new programming methodology and toolset for heterogeneous many-core platforms that build upon previous works on processor virtualization and component-based software engineering. We dedicate this section to the presentation of these technologies in order to make this paper as self-contained as possible.

### 2.1 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 of 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, i.e. the *front-end*, consists of compiling the program sources into a processor-independent bytecode representation. The second part, i.e. the *back-end*, provides an execution platform that can 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- 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 workstations and debugging them on the platform. Second, the same bytecode can be loaded on a heterogeneous MPSoC, and the processor on which it will be executed can be chosen at runtime.

The main technologies in use today for processor virtualization are the Java bytecode, the Low Level Virtual Machine (LLVM) and the Common Language Infrastructure (CLI).

The Java framework [24] defines a bytecode-based virtual machine and a standard library for the Java language [19]. The lightweight version of Java, namely Java Micro Edition (ME), has been widely accepted in heterogeneous embedded systems in order to provide complementary capabilities, like games for mobile phones or TV guides for set-top-boxes. However, programs written in this language incur a significant performance overhead at run time because of many high level services like object-orientation, garbage collection, multi-threading, etc. Therefore, the use of Java remains constrained to the host processor for the non-critical part of the application.

The Low Level Virtual Machine Compiler Infrastructure (LLVM) [22] provides a very versatile and open compiler architecture for implementing optimizations across the entire lifetime of programs, i.e. at static compilation and link time as well as dynamically through just-in-time code generation. LLVM provides an intermediate representation which is intended to be processor independent, input language independent and well suited for optimized code generation. However, the LLVM format has not been standardized, and, as such, is subject to changes.

The Common Language Infrastructure (CLI) [17, 21] is an international standard that defines a rich virtualization environment for the execution of applications written in multiple languages. Beside the .NET Framework [32] and the .NET Micro Framework [27] provided by Microsoft, there exist several open-source programming environments based on CLI. Mono [28] providing the necessary software to develop and run .NET applications on various operating systems. DotGNU Portable.NET [15] is another implementation of the CLI, that includes everything needed to compile and run C# and C applications that use the base class libraries. ILDJIT [6] is a research compiler that takes advantage of multi-cores to process several just-in-time compilations in parallel. Finally, GCC4CLI [10, 11] is a C Compiler that was designed for the generation of efficient CLI code for embedded systems. Previous studies have shown that the size of CLI code generated by GCC4CLI is close to the size of native binaries for dense instruction sets, namely the x86 and Thumb [12] as well as embedded processors like SH-4 [10].

### 2.2 Component-Based Software Engineering

Although the foundations of composing the software systems by assembling components appeared very early [26], component-based programming has been widely accepted as a new programming paradigm in the last decade for succeeding the object-oriented programming [39]. 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 architecture description languages (ADL).

Many component models have been used in general purpose computing during the last decade for improving program modularity and managing software distribution. Among them, the most adopted ones include the Component Object Model [2] family (COM, COM+, DCOM) from Microsoft, the CORBA Component Model [33] (CCM) from OMG, the Enterprise Java Beans (EJB) [14] from Sun Microsystems and the Open Services Gateway Initiative [35] 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) 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 [38] (RTSC) from Texas Instruments provides an ADL toolset based on JavaScript and a packaging format for building modular system software from component libraries. OpenMAX [34] 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 [37] 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. Recently, Khronos published the specification of the Open Computing Language (OpenCL) [30], which allows the distribution of C functions, called kernels, across many-core graphical processing units. Nevertheless, OpenCL is mainly focused on the requirements of imaging applications and only deals with homogeneous computing platforms. Finally, Cecilia [7] is a C-based implementation of the Fractal Component Model [4] and provides a deployment environment for distributing multimedia components on heterogeneous MPSoC platforms. In addition, Cecilia provides an extensible ADL toolset [23] that allows the integration of new code generation features by third party developers.

## 2.3 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 in the context of 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 many-cores, (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 runtime dependencies.

## 3 VC4SoC

Our proposal, namely Virtual Components for SoC (VC4SoC), consists of a programming toolset that enables component-based programming in C for building target-independent component libraries that can be deployed on heterogeneous many-core platforms. We start this section with an overview of 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.

### 3.1 Compilation Flow

As depicted in Figure 1, the development flow with VC4SoC is separated into two parts, that we call *front-end* and *back-end*. The *front-end* is about compiling source component libraries to binary component libraries. It executes on a workstation. The *back-end* is about the composition of applications from binary component libraries. It executes either on a workstation for static mapping of applications on a target platform, or on-board to enjoy dynamic mapping at deployment time.

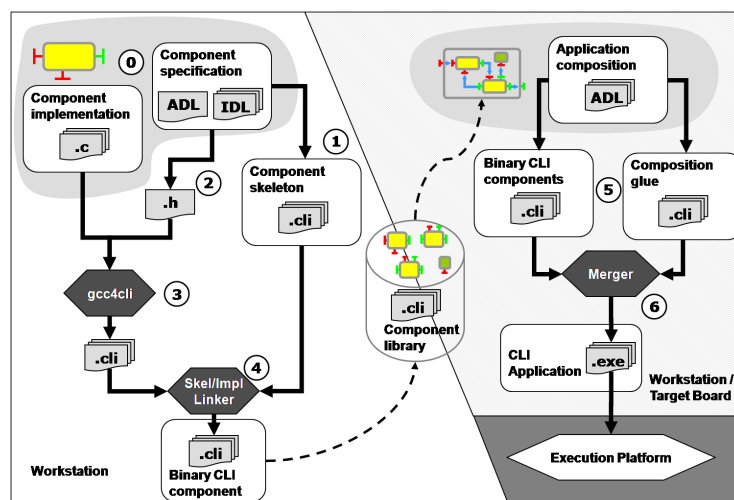


Figure 1: Overview of the compilation flow.

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

verifications to be done and part of the interaction code to be generated. Second, an *Architecture Description Language* (ADL) is used to specify 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. This way, some static verifications can be done and the glue code required for assembling the components can be generated.

These specification files are used for generating two separate files. To start with, 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 structure representing the component without the implementation of its interfaces. The implementation of the component is provided by the programmer in C language using a lightweight programming model based on standard pre-processing macros. These macros are used for accessing component's members such as client interfaces, private data fields, etc. This implementation code 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 using a C-to-CLI compiler (3). Finally, a specific linker tool, described in Section 3.2.3 is used for merging 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. In this case, the glue code may be generated directly on the target platform in CLI format, using a lightweight bytecode manipulation tool such as Cecil [29]. Finally, a *merger* tool may be used for gathering the components that will be deployed on the same processor in order to optimize them for the target execution environment (6).

## 3.2 Component model

VC4SoC is based on the Cecilia Component Framework [7], a lightweight implementation of the Fractal Component Model [3, 5] dedicated to the development of embedded applications and systems. This section starts with the presentation of VC4SoC components' CLI-based binary layout. Then it describes the programming model and the linker tool that is designed for encapsulating standard C programs into these components.

### 3.2.1 Binary layout

The VC4SoC provides a cost-effective implementation of components based on unmanaged CLI structures. As depicted in Figure 2, a component at runtime 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* (1), 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* (2) for each of its *server interfaces*. These *interface structures* contain pointers giving access to the implementation (3) of the methods implemented by the component.

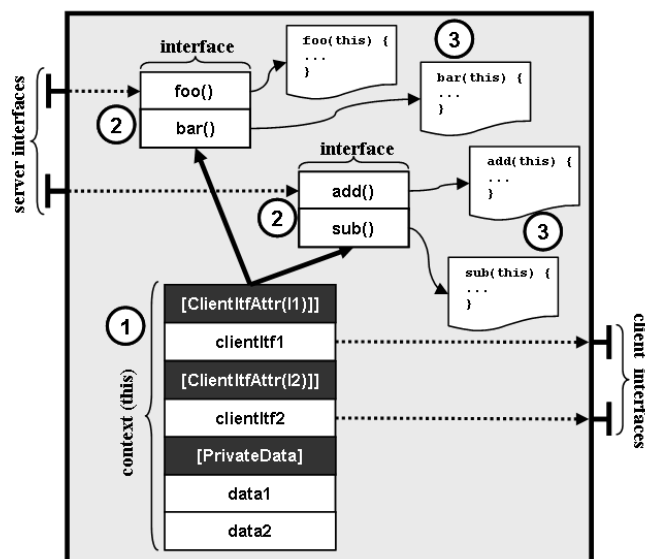


Figure 2: Binary structure of a component in CLI.

In addition to the above functional fields, the *component context* structure is annotated with CLI metadata attributes<sup>2</sup> giving supplementary information about the component's architecture (interface names, component name, etc.). This information is required by the

<sup>2</sup>CLI metadata attributes are represented as gray boxes making part of the *component context* structure.

```

#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)

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

*back-end* of the VC4SoC infrastructure for composing applications from binary component libraries. This metadata can be removed at runtime 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. Indeed, it is expected to be even less than the latter since the former requires only unmanaged CLI structures<sup>3</sup>.

### 3.2.2 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.

Figure 3-b illustrates the implementation of a simple component printing a message using a client *printer* interface. `PRIVATE_DATA` and `DATA` macros are 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 interface to be invoked.

<sup>3</sup>Unmanaged CLI structures are less expensive than managed objects in terms of performance since they do not imply any runtime support.

The implementation code depicted in 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 Figure 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. In order to avoid name clashes, the `METHOD` macro mangles the name of the implemented method with the name of the interface and the name of the component it belongs to.

Careful readers may have noticed that the type definitions in the above header file do not match the binary layout presented in Figure 2. In particular, (i) the *component context* does not inherit from the server interface structures and (ii) 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, so that it can be compiled with a standard C compiler. 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, 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 Figure 3-c<sup>4</sup>, the component skeleton uses these features for defining the *component context* structure matching exactly the binary layout presented in Section 3.2.1. 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.

### 3.2.3 Skeleton/implementation linker

As described previously, during the first part of the compilation flow, the hand-written *implementation* in C gets 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 file. This operation is trivial 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 trivial as well since the methods to be copied are annotated using specific attributes (see the `cli` attribute in Figure 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. The singleton content definition that uses a global context variable `_this` (Figure 3-b) is turned into standard CLI structure containing method implementations from which multiple instances can be created at runtime.
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 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 Mono.Cecil [29]. Such a CLI manipulation tool can also be used in the *back-end* of the compilation flow in order to implement optimizations such as the removal of CLI attributes and the merge of multiple components into one.

## 4 Discussion

This section discusses about the results, limitations and the perspectives of the work presented in this paper.

### 4.1 Results

We have implemented the VC4SoC compiler as an extension of the Fractal ADL Compiler [23]. 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 execution platform (e.g. Mono [28]).

While our proof-of-concept prototype is at an early stage of development, it already handles many toy applications that are not presented in the paper. Specific 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 reported for the technologies that we combined have separately demonstrated that they have acceptable overheads within the context of embedded systems. In particular, [36]<sup>5</sup> has shown that the fine-grained component-based re-engineering

<sup>4</sup>Note that, for readability purposes, we used C# syntax to illustrate the generated skeleton. The actual skeleton is directly generated in CLI.

<sup>5</sup>The performance results presented in [36] and [18] are valid for Cecilia. Indeed Cecilia is a successor of the Think component model.

of an H.264 video decoder resulted in 1.5% overhead in execution time and in 7% overhead in memory footprint while easing notably the distribution of the decoder on multiple processors. Moreover, [18] has shown that modular design with component-based programming may also result in better performance by designing on-demand software systems. The performance results reported in [36] and [18] are similar to the penalty of object-oriented languages. The memory cost reported in [36] is mainly due to the architecture information that is interlaced in components' binary layout. As presented in Section 3.2.1, our approach uses CLI annotations for that purpose. Therefore, these annotations can be removed by the *back-end* if the application does not need them at run-time. Finally, [10, 11] report that efficient CLI code can be generated from the C language and the resulting binaries have competitive sizes with the native binaries of many processors [10, 12].

It is also important to note that the performance of applications developed with VC4SoC is tightly coupled to the performance of the target execution platform. Previous studies [16] 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.

Those limited performance penalties have to be contrasted with the significant gain brought by our approach in productivity and the additional flexibility.

## 4.2 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 [11]) 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 believe 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 would not be 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 [17]. We believe that, in any case, this approach is much better in terms of software engineering, i.e. code readability, portability, maintenance, etc.

## 4.3 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.

### 4.3.1 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 lifecycle into two phases as presented in Section 3.1 is key to breaking 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 runtime from a given component library.

### 4.3.2 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 limits the mapping flexibility.

VC4SoC components provide two features that are key to solve this issue. First, binary components can be introspected to get access to their interface specifications. Second, CLI bytecode can be directly generated on-board using bytecode manipulation tools such as Cecil [29]. 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.

## 5 Conclusion

This paper proposes a new programming methodology and toolset to improve the software development practice on heterogeneous many-core platforms. Our approach has its roots in two well established technologies, namely component-based software engineering in C 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 environment by offering them a single virtual target for all cores present on the platform. This greatly simplifies the software engineering process and reduces the burden, thus the cost, associated with software development for multiple and heterogeneous targets. We believe that this combination of technologies provides a promising groundwork to address the code generation challenges of the future systems 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 runtime, 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.

## 6 Acknowledgments

We would like to thank Philippe Guillaume and Matthieu Leclercq for their numerous contributions to our component based software engineering framework. We would also like to thank Benoît Dupont de Dinechin who was specially helpful in promoting the virtualization technology and making it real.

## 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. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.
- [2] D. Box. *Essential COM*. Addison Wesley - Object Technology Series, 1998.
- [3] 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*, 36(11-12):1257–1284, 2006.
- [4] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Composition Framework, *The ObjectWeb Consortium - Interface Specification*. <http://www.objectweb.org>, June, 2002.
- [5] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model, v2, 2003.
- [6] S. Campanoni, G. Agosta, and S. Crespi Reghizzi. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.*, 43(4):11–20, 2008.
- [7] Cecilia web site. <http://fractal.objectweb.org/cecilia-site/current/>.
- [8] Computing Systems Consultation Meeting. *Research Challenges for Computing Systems – ICT Workprogramme 2009-2010*. European Commission – Information Society and Media, Braga, Portugal, Nov. 2007.
- [9] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: a DSL approach to specifying streaming applications. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 1–17, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [10] M. Cornero, R. Costa, R. Fernández Pascual, A. C. 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*, volume 4917 of *LNCS*, pages 130–144, Göteborg, Sweden, Jan. 2008. Springer Verlag.
- [11] R. Costa, A. Ornstein, and E. Rohou. CLI Back-End in GCC. In *GCC Developers' Summit*, pages 111–116, Ottawa, Canada, July 2007.
- [12] R. Costa and E. Rohou. Comparing the size of .NET applications with native code. In P. Eles, A. Jantsch, and R. A. Bergamaschi, editors, *CODES+ISSS*, pages 99–104. ACM, 2005.
- [13] K. De Bosschere, W. Luk, X. Martorell, N. Navarro, M. O'Boyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Sez nec, P. Stenström, and O. Temam. *High-Performance Embedded Architecture and Compilation Roadmap*, volume 4050/2007 of *Lecture Notes in Computing Science*, chapter 1, pages 5–29. Springer Berlin / Heidelberg, 2007.
- [14] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans, version 3.0. Technical report, Sun Microsystems, 2006.
- [15] DotGNU project. <http://dotgnu.org>.

- [16] 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 (Euro-Par 2008)*, Las Palmas de Gran Canaria, Spain, Aug. 2008.
- [17] ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *Common Language Infrastructure (CLI) Partitions I to IV*, 4th edition, June 2006.
- [18] 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.
- [19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, June 2000.
- [20] Intel Corporation. Intel Concurrent Collections for C/C++, Reference Manual, 2008.
- [21] International Organization for Standardization and International Electrotechnical Commission. *International Standard ISO/IEC 23271:2006 - Common Language Infrastructure (CLI), Partitions I to VI*, 2nd edition.
- [22] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
- [23] 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.
- [24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, Apr. 1999.
- [25] T. Mattson and M. Wrinn. Parallel programming: can we please get it right this time? In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 7–11, New York, NY, USA, 2008. ACM.
- [26] M. D. McIlroy. Mass produced software components. In *Proceedings of NATO Software Engineering Conference*, pages 138–155, 1968.
- [27] Microsoft. Introducing the .NET Micro Framework. Product Positioning and Technology White Paper, Sept. 2007.
- [28] The Mono Project. <http://www.mono-project.com>.
- [29] Cecil Library. <http://www.mono-project.com/Cecil>.
- [30] A. Munshi. The OpenCL specification version 1.0. Technical report, Khronos OpenCL Working Group, 2009.04.02.
- [31] P. Murthy. Parallel computing with x10. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 5–6, New York, NY, USA, 2008. ACM.
- [32] Microsoft .NET. <http://www.microsoft.com/.NET>.
- [33] OMG. CORBA 3.0 New Components Chapters. Technical Report OMG TC Document otc/2001-11-03, Object Management Group, 2001.
- [34] OpenMAX Development Layer API Specification, Version 1.0.1. Technical report, The Khronos Group Inc, June 2006.
- [35] Open Services Gateway Initiative, OSGi service gateway specification, Release 2, October 2001. <http://www.osgi.org>.
- [36] 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 *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia, ESTMedia 2005*, pages 95–100. IEEE Computer Society, 2005.
- [37] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonard, B. Lavigueur, and D. Lo. Distributed object models for multi-processor SoC's, with application to low-power multimedia wireless systems. In G. G. E. Gielen, editor, *DATE*, pages 482–487. European Design and Automation Association, Leuven, Belgium, 2006.
- [38] Real-Time Software Components. <http://wiki.eclipse.org/DSDP/RTSC>.
- [39] C. Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
- [40] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.