2009 International Conference on Parallel Processing Workshops

Towards a Component-based Observation of MPSoC

Carlos Prada-Rojas, Vania Marangozova-Martin,
Kiril Georgiev*, Jean-François Méhaut
Laboratoire d'Informatique de Grenoble
51, Av. J. Kuntzmann, 38330 Montbonnot, France
Email: firstName.lastName@imag.fr

Miguel Santana
STMicroelectronics
850, rue Jean Monnet, 38921 Crolles Cedex, France
Email: Miguel.Santana@st.com

***Abstract*—Motivated by the increasing heterogeneity and complexity of MPSoC systems, we propose a component-based generic approach for MPSoC observation. We show that components help in observing all software levels from system to application. We present the EMBera prototype and relate our experience in implementing it on two different platforms: a Linux-based 16-core SMP machine and a 5-core embedded system developed by STMicroelectronics.**

***Index Terms*—MPSoC; observation; component; parallel programming; NUMA.**

I. Introduction

The design of multi-processor systems-on-chip (MPSoC) is a notoriously complex issue[1]. Manufacturers have to integrate new hardware technologies, to develop new system software and to provide new sophisticated functions in a very short time to market.

A typical MPSoC commercialization involves the porting of system software to the new architecture, the production of dedicated development tools and the platform-specific implementation of MPSoC applications. This platform-dedicated process is however not adapted to future MPSoC which follow current trends in processor development and will integrate dozens and even hundreds of computing cores in various hardware architectures.

To face the challenge of parallel multi-core architectures, MPSoC need new programming models and development tools. To ensure rapid software development and efficient software tuning, MPSoC need new solutions for debugging and performances evaluation both of which are to be based on well-suited MPSoC observation.

Current techniques for MPSoC observation consist in gathering execution traces mostly from hardware and the operating system. Because these solutions are closely related to the underlying platform, they offer a poor extensibility to new architectures and programming strategies. In order to overcome the lack of extensibility, we propose to use software components.

Indeed, software components have proven to be a good solution for reusing and organizing application code [2]. They are largely used in the software engineering domain and have been accepted in the distributed systems area [3][4]. The MPSoC domain has started using components but mostly in application development [5]. Using components for observation purposes will enable the isolation of low-level system concerns from application level issues. This separation of concerns may be used for observation of the application in a more comprehensive way.

In this paper, we investigate the use of components for observing MPSoC applications. We propose a component model for applications and show that it can be used for multi-level observation. We relate our experience in implementing this model on two different platforms, discuss the problems of implementation as well as what the basic observation functions of a system should be.

This paper is organized as follows. Section II presents related work. Section III introduce EMBera which is our proposition for observing MPSoC using components. Sections IV and V describe our experience in implementing EMBera on two different platforms. Our conclusions about the use of components and the needed observation functions are presented in Section VI.

II. Related Work

The observation of applications is a problem which has been already addressed by numerous domains. In this section we focus on embedded, parallel and software component systems. Embedded systems are considered as they define our working context. Parallel systems are important to study as, with the apparition of multi-core architectures, MPSoC become "de facto" parallel systems. Finally, we consider components as they provide a high-level representation of a target system.

Observation on MPSoC: The historically concurrent development of SoCs' software and hardware has resulted in the production of software that is specific to the underlying hardware. Indeed, SoC software is usually low-level (drivers, operating systems) and in charge of the management of a proprietary hardware. As a consequence, the tools developed

1530-2016/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPPW.2009.76

for SoC platform observation are also proprietary and low-level. They mostly give information about hardware state (memory dumps, CPU register values) and kernel events (interruptions, function calls). They usually do not provide information about the application layer and even if they do, there is no mapping between application operations and lower-level observation data. Examples of typical SoC observation tools are KPTrace [6] and OS21 Activity Viewer, both developed by STMicroelectronics. The first tool operates on a Linux based system [7] while the latter works on OS21, an in-house real-time operating system. Another example is the SpyKer product [8] proposed by LynuxWorks.

Observation on Parallel Systems: There are some very efficient observation tools in parallel systems. Shared memory systems, for example, use the thread programming model and are provided with thread observation tools [9][10]. In the same time, there are tools [11][12] for monitoring distributed-memory parallel applications written in MPI [13] or OpenMP [14]. However, as MPSoC are not likely to become dedicated to a given parallel programming model, these existing tools cannot be applied ”as is”.

Observation on Component-based Systems: A different approach to observation is proposed in component-based software systems. Unlike SoC systems, observation is mostly focused on high-level software layers like end-user applications and component-oriented middleware. It typically covers the component architecture and components' interactions. The Fractal component model [2], for example, can detail the set of executing components and the existing bindings between them. It can also trace component creations and communications. Similarly, OpenCCM [15], an open-source implementation of the CORBA Component Model, uses interceptors in order to capture method invocation, and thus, monitor component creations and communications. The same approach is applied to the implementations of the EJB model [3]. Component observation at the application level has an important advantage which is to be independent of the underlying system. However, it is unfortunately unrelated to low-level performance metrics which are crucial for embedded system development.

To bridge the gap between components and MPSoC it is possible to use component-based operating systems. However, existing implementations either do not target MPSoC architectures [16][17], or do not consider the aspect of observation. [18]. The only project applying components to MPSoC observation we are aware of is the Nomadik Multiprocessing Framework project [5] of STMicroelectronics in which our work is to be integrated.

III. The EMBera Observation Model

The major motivation behind EMBera is to provide an observation solution for embedded systems so that:

- it can be used to observe different types of embedded applications (i.e. application-independent).
- it can be used to observe different levels related to the execution of an embedded application.
- it can be used on different MPSoC hardware platforms (i.e. platform-independent).
- it can be configured to serve a specific observation context.

In other terms, our main objective is to be able to define observation in generic terms and yet be able to efficiently observe specific embedded hardware and software. We will focus on the way to define observation functions separately from the embedded platform. Then, we will use components as they appear to be a successful solution to the problem of separation of concerns.

The EMBera model is based on the Fractal component model [2]. We have chosen Fractal since it is a general component model that is system and language independent. Indeed, it can be used at the system level, as well as middleware or application levels and it can be implemented in Java, C or other programming languages. Another major advantage of Fractal is that it is already used at STMicroelectronics which defines our working context.

A. The EMBera Component Model

An EMBera application is composed of a number of interconnected components. A component is a software entity with a well-defined functionality. A part of this functionality can be visible to other components, in this case the component defines provided interfaces. Some components may depend on this functionality, in that case they define required interfaces. Connections between components are established by linking required and provided interfaces.

The components in EMBera are active entities and each component has its own execution flow. This choice follows the current practice for MPSoC applications in which multiple treatments are executed on different processor units.

EMBera components provide a predefined interface for component control. The control operations include component creation, component interconnection and component life-cycle management (launching and termination).

B. The Motion-JPEG Decoder Application Example

To illustrate the EMBera model, let us consider a typical MPSoC application: a video codec which decodes a stream of independent and individually encoded JPEG images[1]. The decoding is done by dividing each individual image in smaller blocks. Each block is decoded mainly by applying a Huffman algorithm, a pixel reordering and the Inverse Discrete Cosine Transformation (IDCT). Then, all the blocks are reordered in order to reconstitute original images.

[1]Implemented for [19] in the scope of the cycle-accurate simulation platform.

For computing independent data in parallel, the MJPEG decoder code can be divided into three parts. The first part is responsible for file management, Huffman decoding and pixel reordering. A second part executes the IDCT treatment, either in sequence on all blocks, or in parallel by separating the blocks. Finally, the third part deals with the reordering of blocs. If we set these parts onto components, we obtain the application on Figure 1. The connections between components are used to manage the data flow.

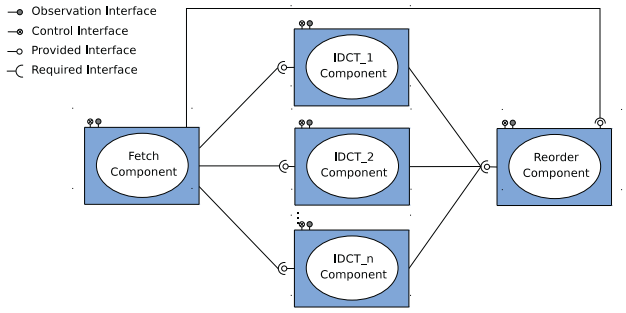


Fig. 1. Componentized MJPEG Decoder Application

C. Observation in EMBera

We have decided to explicitly model the observation in EMBera. For this purpose, we have defined a new control interface dedicated to observation.

We consider that MPSoC observation has to take into account at least three levels: the system, the middleware and the application level. The observation interface may provide functions related to each level such as memory and system time, communication time, and application structure (e.g. the component structure). However, the exact information to be provided by this interface is still to be defined.

The information obtained, accessible through the observation interface, is gathered and analyzed by a new component connected to the observation interfaces. We have named it the observer component.

D. Implementation of the EMBera Model

We have implemented the EMBera model on two different platforms: a 16-core SMP Linux system and a 5-Processor STMicroelectronics MPSoC. The former platform is a standard x86 multiprocessor architecture, while the latter is an MPSoC currently used in STMicroelectronics products.

The main propose of these implementations is to validate that a component model provides the means for observing embedded applications at different levels. Another objective is to try to identify the basic observation functions to be provided by default at each level. These implementations do not address the intrusiveness problem due to the component implementation or to the data collection.

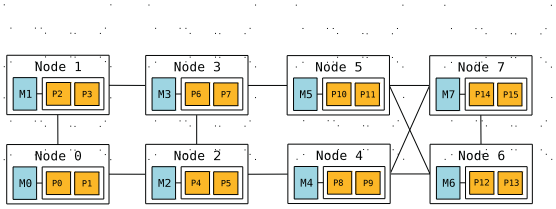


Fig. 2. 16-core Symmetric Multiprocessor Platform.

IV. Implementation of EMBera on SMP - Linux

The 16-core platform is a Symmetric Multiprocessor eight dual core AMD Opteron 2.2 GHz and 2 MB of cache memory for each processor. It is organized in eight nodes and has in total 32 GB of main memory (4GB of local memory). Each node has three connections to communicate with other nodes. This platform uses a Linux kernel 2.6, providing native C compilation and POSIX thread support [20].

This platform follows a NUMA (Non-Uniform Memory Access) memory organization. A NUMA platform is a multiprocessor system in which the processing elements are served by multiple memory levels, physically distributed through the platform. Such distributed memory is seen by the application as a single shared memory [21]. However, the access time to the distributed memory changes depending on the distance between the processor and the memory.

A. Implementation of the EMBera Component Model

The implementation of EMBera is done in the C language as it is the ”de facto” standard for embedded software due to its performance and to legacy reasons [22].

An EMBera application is a Linux user process. A component is composed of a data structure and a POSIX thread. The thread belongs to the Linux user process and provides an execution support for the code inside the component.

The communication between components is based on simple one way asynchronous message-oriented mechanism providing the `send` and `receive` primitives. A provided interface is represented by a FIFO data structure, we have named mailbox through which it receives messages. A required interface is a pointer to a provided interface (mailbox) which is used to send messages. A connection is established by setting the pointer on the required interface to a given provided interface.

The deployment of any EMBera application is carried out by explicitly invoking control functions into the `main` application function.

B. Implementation of the Observation Interface

The observation interface is implemented as a couple of interfaces (one provided and one required) and a set of observation functions.

This couple of interfaces for the observation is created by default on any EMBera component. A component may thus

receive messages requesting observation information (using the provided interface) and return the requested information (using the required interface).

The set of observation functions concerns functions for collecting execution data from different software levels. The current implementation addresses three levels of observation: the operating system, the middleware, and the application itself. The operating system is the Linux system software which directly manages the SMP platform. Most of the information related to the platform and resources utilization can be retrieved or inferred from this level. The middleware concerns the EMBera communication primitives, the application level being the component structure and the code inside EMBera components.

The observation information provided is obtained by implementing the observation functions into the EMBera component implementation without modifying the application code. We will now describe the functions currently implemented for observing each level.

Operating System: We have currently gathered information about the execution time and the memory occupation. For obtaining the execution time, we have calculated the time elapsed between the start and the termination of a component. The time has been measured by using the `getTimeOfDay` system function.

For obtaining the component memory, we have calculated the memory allocated for the component thread and the size of memory allocated for all the component provided interfaces and related structures. These measures have been gathered by using `pthread_attr_getstacksize` and `sizeof` functions respectively.

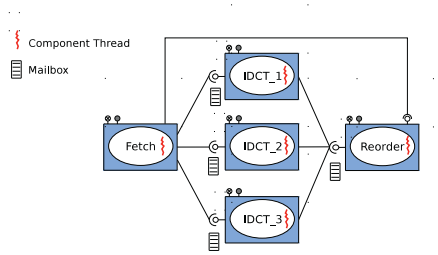
Middleware: We have obtained information about the execution time of `send` and the `receive` operations by instrumenting the corresponding primitives. The time stamping is also supported by the `getTimeOfDay` system function.

Application: The information we have collected is about the component structure and the total number of communication operations performed. The former consists in listing provided and required interfaces of the components, while the latter is achieved by adding counters to `send` and `receive` primitives and associating them to components.

C. Implementation of The Motion-JPEG Decoder

Figure 3(a) depicts the architecture of the MJPEG EMBera application implemented with five components: one Fetch, three parallel IDCT and one Reorder component. Figure 3(b), shows the `main` application function, in which each one of the five components and its interfaces are instantiated. Then, this function specifies the connections between all the components.

The MJPEG application is executed on two different input files containing 578 and 3000 JPEG images respectively. The



(a) Component Architecture

```
component *fetch = createComponent (10, "fetch", fetch_function);
interface *fetchReorder = createInterface(fetch, "fetchReorder", "client");
interface *fetchIDCT1 = createInterface(fetch, "fetchIDCT1",  "client");
...
component *idct1 = createComponent(20, "idct1", idct_function);
interface *_fetchIdct1 = createInterface(idct1, "_fetchIdct1", "server");
interface *idct1Reorder = createInterface(idct1, "idct0Reorder", "client");
...
component *reorder = createComponent (30, "reorder", reorder_function);
interface *_fetchReorder = createInterface (reorder, "_fetchReorder", "server");
interface *_idctReorder = createInterface (reorder, "_idctReorder", "server");

componentConnect (fetch, fetchReorder, reorder, _fetchReorder);
componentConnect (fetch, fetchIDCT1, idct1, _fetchIDCT1);
...
componentConnect (idct1, idct1Reorder, reorder, _idctReorder);
...
```

(b) Deployment Program

Fig. 3. EMBera MJPEG Architecture and Deployment

dimensions of each single image are the same in both cases.

D. Analysis of Observation Data

Let us look into more detail the collected information data.

Operating System: The data in table I gives the components' execution times and the component memory initially allocated.

Component	Time_{578} (μs)	Time_{3000} (μs)	Mem (kB)
Fetch	4 084	20 088	8 392
IDCTx	4 084	20 218	10 850
Reorder	4 086	21 538	13 308

TABLE I
MJPEG COMPONENTS EXECUTION TIME AND MEMORY ALLOCATED

We observe in both execution cases that having three IDCT components computing in parallel balances the execution times of the three parts of the MJPEG application.

For this experience, the memory values obtained for a Linux thread stack correspond to 8 392 kb. The memory allocated to the Fetch component memory corresponds to this value, therefore, the component does not instantiate any provided interface. Higher memory values for IDCTs and Reorder represent their provided interfaces and can thus be used to evaluate the memory consumption of our component implementation.

Communication: Figure 4 presents the evolution of `send` execution time when the message size increases. The execution

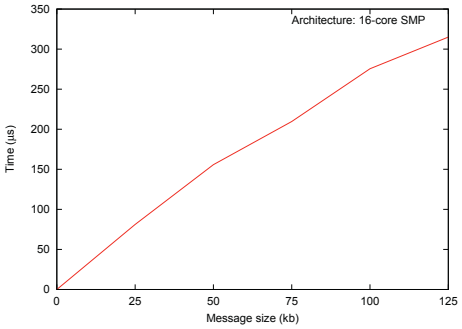


Fig. 4. `send` Primitives Execution Time.

time values obtained show that the time spent for sending a message increases almost linearly with the size of the message. This shows that, in the considered execution cases, the `send` operation depends only on the message size and not of the communication mechanism, nor the SMP platform's architecture.

Application: One first information obtained gives the number of communication operations (table II).

Component	$send_{578}$	$receive_{578}$	$send_{3000}$	$receive_{3000}$
Fetch	10 386	0	53 982	0
IDCTx	3 462	3 462	17 994	17 994
Reorder	0	10 386	0	53 982

TABLE II
MJPEG COMPONENTS COMMUNICATION OPERATIONS PERFORMED

The values in this table indicate that the Fetch component sends messages but it does not execute any `receive` operation. The IDCT components receive and send the same amount of messages. Finally, the Reorder component receives the same number of messages as initially sent by Fetch. If we do not have access to the internal code of components, this information is useful to infer the functioning of the application. The Fetch component takes an input file, prepares a set messages and sends one third of this set to each IDCT. Each IDCT treats the received message and sends a result to the Reorder component. The Reorder component receives all results from the three IDCTs.

```
Interfaces component [IDCT_1]
------------------------------
[Interface]     [Type]
introspection   provided
_fetchIdct1     provided
introspection   required
idctReorder     required
```

Fig. 5. Interfaces for Component IDCT_1

Another information obtained is related to the components' structure. Figure 5 shows that IDCT1 component has four interfaces: the two observation interfaces (one provided and one required), the provided interface for the Fetch component and the required interface to connect to the Reorder component. This observation can provide valuable information for applications whose configuration changes dynamically.

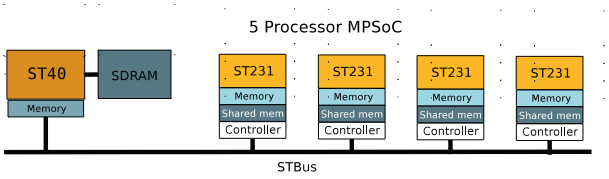


Fig. 6. STi7200 Platform

The information obtained by observing the MJPEG application through the observation interface ensures a better understanding of the application behavior and therefore helps to find potential performance improvements. For example, the execution times indicate that the application is well load-balanced for the JPEG input size but if that size changes, the execution times could cause a bottleneck on the IDCT components.

V. IMPLEMENTATION OF EMBERA ON MPSOC - OS21

The STi7200 MPSoC platform (Figure 6) is composed of one 450 Mhz general purpose RISC CPU (ST40) and four 400 Mhz accelerators (ST231). The ST40 CPU has access to the total on-chip memory including one big external block of 2 GB SDRAM memory. Each ST231 CPU has access to a block of local data and control memory.

The ST231 and ST40 CPUs communicate using one shared block of memory associated with one interruption controller. The chip can be programmed by using STMicroelectronics implementation of standard ANSI C. It is supplied with a complete toolset including among others optimized compilers, assemblers, linkers and observation tools. As ST40 and ST231 processors have different instruction sets, each has its own toolset.

STi7200 processors run OS21: a lightweight, real-time multitasking operating system (RTOS). The OS21 RTOS provides portable APIs to handle tasks, memory, interrupts, exceptions, synchronization and time management. The OS21 tasks behave like processes and communicate via a specific middleware, called EMBX, developed by STMicroelectronics. This middleware manages shared memory regions accessible by several CPUs. These memory regions are called distributed objects and are accessed by dedicated `EMBX_Send` and `EMBX_Receive` functions. The `EMBX_Send` is an asynchronous operation corresponding to a write operation on the distributed object. The `EMBX_Receive` is a synchronous operation corresponding to a read operation on the distributed object.

A. Implementation of the EMBera Component Model

An EMBera application is a set of OS21 tasks, each task representing a component. The current implementation

supports one component per CPU and thus avoids dealing with the low-level multi-tasking OS21 support.

A component's provided interface is represented by a distributed object. A component's required interface corresponds to a pointer towards a distributed object. A connection between both interfaces is established using EMBX primitives managing distributed objects.

When a component needs to communicate through a required interface, it executes `EMBX_Send` and thus updates the corresponding distributed object. As the distributed object represents a provided interface, the component providing interface needs to execute `EMBX_Receive` in order to complete the communication.

The deployment of an EMBera application on the STi7200 platform consists in loading one binary code per CPU which is performed by using STMicroelectronics proprietary devices and software tools. Each binary code contains a `main` function which creates, connects, starts and stops the component.

B. Implementation of the Observation Interface

As shown in the Linux EMBera implementation, the observation is applied to three software levels: the system, the middleware and the user application. We will discuss the observation at the system level and at the middleware level since the user application level observation is identical to the Linux implementation.

Operating System: At the system level, our objective is to observe the system memory utilization and the component task execution time. When the OS21 initializes, the component task is created and starts . We can observe the task execution time by using an OS21 supplied `task_time` system function.

The system memory used by an EMBera component is related to the local memory for the task and the SDRAM memory for the distributed objects. The observation of the local memory is carried out by OS21 functions. Those functions provide the task's memory size and the amount of memory currently used which, in the SDRAM, is equal to the size of all distributed objects. This size value is fixed and gathered at component creation time.

Middleware: At the middleware level the observation mechanism is the same as in the Linux implementation and is based on gathering execution information about the sending and the reception of messages. What differs is that the timestamp is given with the `time_now` OS21 system function which gives the local time on each CPU.

C. Implementation of The Motion-JPEG Decoder

We have deployed the MJPEG application on three (ST40 and two ST231) out of five processors on the STi7200 platform. Indeed, the software toolset provided by STMicroelectronics for our experience supports only three processors.

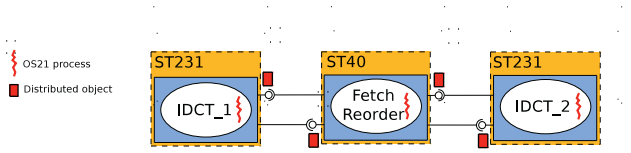


Fig. 7. Componentized MJPEG Decoder Application on STi7200 platform

Figure 7 presents the componentized MJPEG application. We have decided to create a single I/O component by merging the Fetch and the Reorder functionalities in a Fetch-Reorder component. This component is deployed on the general purpose ST40 CPU. It is connected with the two IDCT computation components, each one deployed on one ST231 CPU.

D. Analysis of Observation Data

We will now show and discuss the observation information provided at RTOS level and EMBera middleware level. The information collected at the application level for the OS21 MJPEG implementation does not provide additional information in comparison with the Linux implementation.

RTOS: In table III, we show the overall execution time of the components' tasks and the local memory consumption. On the Linux EMBera implementation, the Fetch and the Reorder components computation times are almost the same as those of the IDCT components. On the OS21 implementation, the Fetch-Reorder component runs ten times slower than the IDCT components. This difference might be due to the ST40 processor which is general purpose and computes slowly the Reorder algorithm.

Component	Time (s)	Mem (kb)
Fetch-Reorder	1173	110
IDCTx	95	85

TABLE III
MJPEG COMPONENTS EXECUTION TIME AND MEMORY ALLOCATED

It is interesting to observe the huge difference between the Linux IDCT component overall execution time (approx. 4 s) and the OS21 IDCT component overall execution time (approx. 100 s). Despite the frequency differences between the SMP platform processors and the STi7200 accelerators, tthis difference is not justified. This problem might occur because we execute the Linux version of the MJPEG code without applying any optimizations.

The 85 KB memory consumption on the IDCT component corresponds to 60 KB for the task data and component structure and 25 KB for one distributed object. The Fetch-Reorder component uses two distributed objects which justifies the 100 KB consumed memory.

The 85 KB allocated memory for the IDCT component is several times smaller than the memory allocated for the Linux IDCT component (8 MB) but is explained by the memory differences between the two architectures (1 MB for MPSoC and 32 GB for Linux).

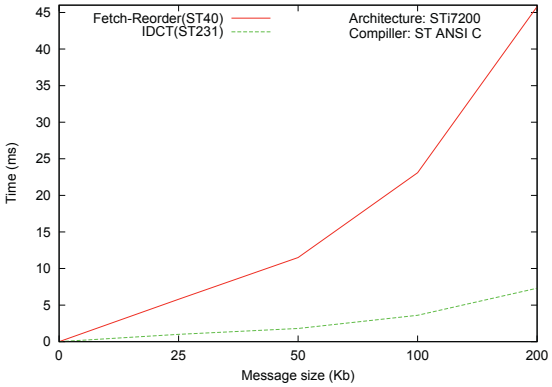


Fig. 8. EMBera `send` execution time over 578 MJPEG images

Middleware: In figure 8, we show the average execution time of the EMBera `send` function for 578 MJPEG images, performed by the Fetch-Reorder component and the IDCT component. We can see, in figure 8, that IDCT component executes the `send` operation faster than the Fetch-Reorder component for the same message size. That difference in the execution time is due to the hardware architecture of the STi7200 platform, which favors the memory operations of the ST231 accelerators. These memory operations are the most time consuming. Indeed, the general purpose ST40 CPU is mainly designed to access peripherals while ST231 accelerators are designed for intensive computing which needs fast memory access.

The message size has a direct consequence on the application performance. Indeed, the performance of the EMBera `send` function is linear for message sizes smaller than 50 KB. Over 50 KB, the send function decreases its performance. Since the performance depends on the size of the message, we can deduce from this observation that OS21 EMBera implementation is well suited for the hardware architecture when the message size is less than 50 KB.

The generic observation information we gathered in this example can be useful for optimizing the communication time between the components. For instance, we can force the Fetch-Dispatch component to send different number of messages, according to the message size, in order to balance the EMBera `send` execution time between the components.

VI. Conclusions and Perspectives

The EMBera model has enabled us to set an application in terms of components. The observation modeled in EMBera provides us with a generic mechanism for obtaining meaningful information about the execution of an MPSoC application. Both implementations demonstrate that the componentized MJPEG application can be observed without modifying its code. Indeed, we are able to observe the application behavior based on the interaction among the application components as well as between the components and other execution levels.

As a matter of fact, we have found interesting to observe at least three execution levels: the operating system, the middleware and the application. At the OS level, we have proposed to implement functions for observing memory and execution times. At the middleware level, we have observed the behavior of communication primitives. Finally, at the application level we have focused on the observation of the use of middleware and on the component structure. According to us, these functions are the basis to observe any MPSoC application.

In the ongoing work, we focus our research on defining and extending EMBera observation functions, for instance, cache misses and the evolution of memory during the execution of a program. For extending observation capabilities, we are working on abstracting operating system observation functions and communication observation metrics from the component model. We will concentrate our future work on what functions should be provided with the observation interface, how to select the events to be observed, how to set the treatments to apply and finally, how to manage multi-level information.

Another important open question is about intrusiveness and data collection treatments. In the long term we are interested in defining an observation mechanism in which it is possible to measure and control the intrusiveness by configuring the data collection treatments.

References

[1] W. Wolf, "The Future of Multiprocessor Systems-on-Chips," in *DAC '04: Proceedings of the 41st annual conference on Design automation*. New York, NY, USA: ACM, 2004, pp. 681–685.

[2] 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 (SP&E)*, vol. 36, no. 11-12, pp. 1257–1284, Sep. 2006, special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[3] "Enterprise JavaBeans Technology," Website, SUN, http://java.sun.com/products/ejb/.

[4] "CORBA Component Model, v4.0," Website, OMG, http://www.omg.org/technology/documents/formal/components.htm.

[5] J.-P. Fassino, "Nomadik Multiprocessing Framework, a Component-based Programming Model for MP-SoC," 7th International Forum on Application-Specific Multi-Processor SoC, June 2007. [Online]. Available: http://www.mpsoc-forum.org/2007/slides/Fassino.pdf

[6] "Dynamic Kernel Tracing with KPTrace," Website, SUN, http://www.stlinux.com/docs/manual/development/advanced_development30.php.

[7] "STLinux," Website, http://www.stlinux.com/drupal/.

[8] "SpyKer," Website, http://www.lynuxworks.com/products/spyker/spyker.php3.

[9] Q. A. Zhao and J. T. Stasko, "Visualizing the Execution of Threads-based Parallel Programs," Georgia Institute of Technology, Tech. Rep. GIT-GVU-95-01, 1995. [Online]. Available: http://hdl.handle.net/1853/3546

[10] "POSIX Thread Trace Toolkit (PTT)," Website, http://nptltracetool.sourceforge.net/.

[11] S. Shende and A. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[12] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP," *Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, 2002.

[13] "Message Passing Interface (MPI)," Website, http://www-unix.mcs.anl.gov/mpi/index.htm.

[14] "The OpenMP Application Program Interface," Website, http://openmp.org/.
[15] "CORBA Component Model, v4.0," Website, ObjectWeb Project, http://openccm.objectweb.org/doc/index.html.
[16] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The Pebble Component-based Operating System," in *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1999, pp. 20–20.
[17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The Flux OSKit: A Substrate for Kernel and Language Research," *Proceedings of 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.
[18] D. Beuche, A. Guerrouat, H. Papajewski, W. Schroder-preikschat, O. Spinczyk, and U. Spinczyk, "The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems," in *In 2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC 99*, 1999, pp. 45–53.
[19] I. Augé, F. Pétrot, F. cois. Donnet, and P. Gomez, "Platform-Based Design From Parallel C Specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 12, pp. 1811–1826, Dec. 2005.
[20] *Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities (Volume 1)*, ser. Information technology—Portable Operating System Interface (POSIX), 1993.
[21] M. Tao, T. Jie, S. Martin, and M. S. A., "Interactive Locality Optimization on NUMA architectures," in *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2003, pp. 133–ff.
[22] E. Rohou, A. Ornstein, and M. Cornero, "Compiling C to CLI for Heterogeneous Multicore SoCs," 5th HiPEAC Industrial Workshop, June 2008.