NASA: A Generic Infrastructure for System-level MP-SoC Design Space Exploration

Zai Jian Jia[1], Andy D. Pimentel[2], Mark Thompson[2], Tomás Bautista[1] and Antonio Núñez[1]

[1]Research Institute for Applied Microelectronics
University of Las Palmas de Gran Canaria
Spain
{cjia, bautista, nunez}@iuma.ulpgc.es

[2]Computer Systems Architecture Group, Informatics Institute
University of Amsterdam
The Netherlands
{A.D.Pimentel, M.Thompson}@uva.nl

***Abstract*— System-level simulation and design space exploration (DSE) are key ingredients for the design of multiprocessor system-on-chip (MP-SoC) based embedded systems. The efforts in this area, however, typically use ad-hoc software infrastructures to facilitate and support the system-level DSE experiments. In this paper, we present a new, generic system-level MP-SoC DSE infrastructure, called NASA (Non Ad-hoc Search Algorithm). This highly modular framework uses well-defined interfaces to easily integrate different system-level simulation tools as well as different combinations of search strategies in a simple plug-and-play fashion. Moreover, NASA deploys a so-called *dimension-oriented* DSE approach, allowing designers to configure the appropriate number of, possibly different, search algorithms to simultaneously co-explore the various design space dimensions. As a result, NASA provides a flexible and re-usable framework for the systematic exploration of the multi-dimensional MP-SoC design space, starting from a set of relatively simple user specifications. To demonstrate the distinct aspects of NASA, we also present several DSE experiments in which we, e.g., compare NASA configurations using a single search algorithm for all design space dimensions to configurations using a separate search algorithm per dimension. These experiments indicate that the latter multi-dimensional co-exploration can find better design points and evaluates a higher diversity of design alternatives as compared to the more traditional approach of using a single search algorithm for all dimensions.**

Keywords- System-level design space exploration; MP-SoC design

I. INTRODUCTION

Today's embedded systems are increasingly based on multi-processor system-on-chip (MP-SoC) architectures. These MP-SoCs often are heterogeneous, consisting of a (potentially large) number of programmable processors for flexible application support as well as dedicated processing elements for achieving power and performance goals [1]. To cope with the design complexity of such systems, system-level design raises the abstraction level of the design process. Design Space Exploration (DSE) is a key ingredient of such system-level design, during which a wide range of design choices need to be explored, especially during the early design stages. Such early DSE is of paramount importance as early design choices heavily influence the success or failure of the final product.

The process of system-level DSE logically consists of two components [2]: 1) the evaluation of a single design point in the design space using e.g. analytical models or simulation, and 2) the search mechanism to systematically travel through the design space. Both DSE components have received significant research attention during the last decades [3,4,5,6]. For example, system-level simulation is a popular method for evaluating single design points [2]. These simulation tools usually operate at a high level of abstraction and are often based on the Y-chart principle [7,8]. The latter means that they decouple application from architecture by recognizing two distinct models for them. An application model – derived from a target application domain – describes the functional behavior of an application in an architecture-independent manner. Subsequently, an architecture model – defined with the application domain in mind – defines architecture resources and captures their performance constraints. Finally, an explicit *mapping step* maps an application model onto an architecture model for co-simulation, after which the system performance can be evaluated quantitatively.

These simulation tools only provide a partial solution since an overall framework is needed to systematically search the design space (using a simulator to evaluate the selected design points). Such a system-level DSE framework should allow for exploring a wide variety of system parameters and design choices, including the number and type of processing elements in the MP-SoC platform, the type of on-chip network, the memory organization, the mapping of application tasks and communications onto architecture resources, scheduling policies, and so on. Evidently, the more details or dimensions are taken into account, the larger the design space that needs to be searched and therefore the more costly the analysis is. Although many DSE approaches, based on a large variety of search techniques, have been proposed, three common factors can be identified in all of them: 1) DSE frameworks are usually targeted to a specific system-level simulation tool (or analytical evaluation method), where each effort typically uses a different kind of simulator. Consequently, it is hard to re-use (elements of) these DSE frameworks; 2) most of the DSE experiments in these efforts focus on a particular (class of) MP-SoC architecture(s); 3) setting up the DSE experiments can be very labor intensive. It is often the case that for every experiment, control scripts need to be (re-)written to manipulate the simulation parameters and configuration files (specifying the

The work described in this paper was funded by the Spanish Ministry for Science and Technology, grant AP2006-02986.

design instance to evaluate) according to the algorithm that searches through the design space. These scripts often are inflexible and hard to re-use for different types of DSE experiments (i.e., assessing different parameters or parameter ranges). To summarize, ad-hoc solutions are dominant in most of these efforts. To the best of our knowledge, there does not exist any generic supporting infrastructure to facilitate and support system-level MP-SoC DSE experiments, and to foster the re-use of software in the context of system-level MP-SoC DSE.

To address this problem, this paper presents a new, generic system-level DSE infrastructure implemented in C++, called NASA (Non Ad-hoc Search Algorithm). Its main goal is to provide a single, common, and modular framework for system-level DSE experiments. NASA allows for incorporating different (existing) system-level simulation tools as well as different combinations of search strategies by means of a simple plug-in mechanism. As a consequence, it provides a flexible and re-usable environment to systematically explore the multidimensional MP-SoC design space, starting from a set of relatively simple user specifications. NASA's output includes information about all explored design points as well as a set of Pareto-optimal design points within the explored design space, which best meet the users constraints such as real-time application constraints, number and types of available components in the platform architecture, costs/area, etc.

The remainder of the paper is organized as follows. In the next section, related work is discussed. Section III provides some preliminaries on system-level DSE. In the Section IV, we provide an overview of NASA, in which its main properties are introduced. Section V subsequently describes various implementation aspects of NASA. In Section VI, we present a range of experimental results, demonstrating NASA's capabilities. Finally, Section VII concludes the paper.

II. RELATED WORK

Performing DSE in a time efficient and accurate way is not a new problem and there exists a large body of related work in this area. Most of the approaches in the embedded systems domain are targeted to the system-level exploration of heterogeneous MP-SoCs [3,4,9,10,11]. Although these efforts are fairly efficient to explore, e.g., the various alternatives for mapping a specific application onto a target MP-SoC architecture, they typically still require significant efforts to (re-)write the scripts that control the evaluation mechanism (analytical model or simulator) during the search through the design space. In reality, this often means that there exists a repetitive effort to build customized scripts for every different kind of DSE experiment.

Several proposals to integrate external design-point evaluation tools in the DSE environment can also be found in literature. In [12], a hierarchical and three-phase DSE methodology is presented. It facilitates the integration of simulators by using a set of tool-dependent interpreters or adapters. Angiolini et al. [13] present a framework that integrates an ASIP tool-chain within a virtual platform to explore a number of axes of the MP-SoC configuration space. However, unlike our work, these efforts do not target integration of external search methods. Moreover, in these efforts, human intervention is still necessary in the feedback loop to the searching and optimization process.

The MultiCube project [14] has similar objectives as the work presented in this paper, but it targets the exploration of the configuration space of homogeneous chip multiprocessors rather than system-level MP-SoC platform DSE. This implies that it has limited or no capabilities to explore different application to architecture mappings, heterogeneous processing elements, different interconnects, etc.

Several efforts also have developed a modular interface-based system-level MP-SoC DSE framework [15,16]. In both cases, different search algorithms can be plugged in, but the resulting DSE is limited in terms of target MP-SoC platforms that can be explored. This last aspect has been addressed in [17], in which the PISA library [18] is used to create a multi-objective DSE framework. Different mapping alternatives can be evaluated for a fixed or flexible platform during the exploring process. However, this work uses analytical models to evaluate design points (so, it cannot incorporate external simulation tools). Also, the chosen representation formats for internal interfaces in [17] are problem specific, which means that they should be modified for each particular problem. In our case, these are dynamically and automatically updated according to an input constraints file. Finally, the kind of platforms generated in [17] is limited to hierarchical bus topologies, while our approach is not restricted to analyzing a particular architecture.

III. PRELIMINARIES

We define the design space as a set of design points for a given set of user constraints. A feasible design point is a system design that meets the user constraints both in terms of mapping and architectural implementation. This means that, for example, only the available number and types of architectural components (processors, memories, networks, etc.) can be used to create a feasible architecture. Moreover, each task of an application has to be mapped onto a processing resource that is capable of performing this task (in term of functionality), while communications between application tasks must be mapped onto communication resources that actually connect the processors onto which the communicating tasks are mapped. If at least one of above conditions is not satisfied, then the resulting design is classified as an infeasible one.

In this context, a design point is a point p in D, where D is the design space defined as:

$$D = d_1 \times d_2 \times \ldots \times d_k \quad (1)$$

Here, d_k refers to the design options in a particular dimension k, and $\times$ is a Cartesian product. Typically, a dimension could represent design decisions that are orthogonal to each other such as mapping application tasks onto processing elements, number and types of processing elements, number and types of memories, network type and/or topology, etc. This way, finding the optimal or near-to optimal design point consists of a multi-dimensional exploration process, searching for the best combination of values in all dimensions of D that optimizes all the imposed objectives (e.g., performance, power, cost, etc.). This, however, means that the size of the design space is equal to the product of the cardinalities of the set d_i, $\forall i=1..k$:

$$|D| = |d_1| \times |d_2| \times \ldots \times |d_k| \quad (2)$$

Consequently, the more dimensions or the larger each d_i, the larger the resulting design space is. To illustrate how large such design spaces can become, we provide an approximation of the size of the design space we address in the results section of this paper. To this end, we use the expression presented in [2] to calculate the size of one dimension, while the size of the multi-dimensional design space would be the product of all dimensions. As a result, the expression of the size of a 3D design space which considers mapping, architectural components and platform topologies, can be roughly approximated as follows:

$$|D| = w \times \left(g^d * e^b * f^c\right) \times b^t \quad (3)$$

Here, t is the number of the tasks of the application, b is the number of processing elements, c is the number of memory elements, d is the number of networks, e, f, g are the types of processing elements, memory elements and networks, respectively, and w represents the number of different platforms or topologies the designer wants to explore. In order to simplify the estimation, the mapping of the application's communication channels onto memory elements has not been taken into account. Using equation (3), $1.3*10^{12}$ alternatives would have to be evaluated for the following (fairly moderate) set of values: $t=7, b=6, c=3, d=4, e=3, f=3, g=3$ and $w=3$. This means that, even when operating at a high level of abstraction, it is infeasible to exhaustively evaluate all these design points. Therefore, heuristic search methods like evolutionary algorithms, simulated annealing, or ant colony algorithms are typically deployed because such search algorithms only need to visit a limited number of design points to provide a convergence path toward an optimal solution.

IV. THE NASA FRAMEWORK

With NASA, we aim to provide a generic infrastructure for performing system-level MP-SoC DSE experiments. To this end, four key properties have been taken into account in the design of NASA:

- *Modularity*. NASA is a highly modular framework in which the interaction between its modules is established by well-defined interfaces, allowing each module to act like an independent black box inside the framework. As a result, different search algorithms, feasibility checkers, and system-level simulators can be easily integrated in a plug-and-play fashion.
- *Flexibility*. According to the designer's needs, different experiments to explore different aspects of the design space can be performed using NASA. For example, as will be explained in more detail later, a key element in NASA is its *hierarchical* DSE approach in which the exploration dimensions are explicitly separated into three levels: platform exploration, architecture exploration, and mapping exploration. Subsequently, the designer can choose to simultaneously explore at all of these levels, or to fix one or more of these levels (e.g., a fixed platform) and to focus the exploration on one or two levels (e.g., mapping exploration only).
- *Reusability*. For a given set of user constraints, NASA is capable of searching the design space in a systematic and automatic way, and to evaluate selected design points. As a result, there is no need in preparing experiment-customized scripts. A new DSE experiment only requires the changing of the constraint values.
- *Extensibility*. Due to the modularity and the well-defined interfaces, new modules or functionalities can be easily plugged in the NASA framework. These new modules could, for instance, handle additional dimensions in the design space, without the need to modify other modules.

The infrastructure of NASA is shown in Figure 1. Essentially, six main modules can be distinguished in the framework: the Search module, Feasibility Checker, Architectural Platform Generator, Translator, Simulator and Evaluator. These modules work as independent black boxes, using well-defined interfaces or text files (represented by the numbered arrows) to interact with each other.

In the Search module, the design space is explored in an iterative fashion. By means of plug-ins, such exploration can be done exhaustively, randomly, or using a heuristic search algorithm such as evolutionary algorithms, simulated annealing, or ant colony algorithms. Moreover, as mentioned before, the DSE approach is hierarchical and currently distinguishes three levels: platform, architectural components and mapping levels. These levels are co-explored simultaneously, possibly using different search algorithms. This means that different (tailored) search algorithms can be used for each level. At the platform level, the platform structure – defining the number of architectural elements and their topological interconnection – is explored. The architectural components level explores the types of architectural components (processor types, memory types, etc.) inside a platform architecture. Finally, at the mapping level, different mappings of application tasks and communications onto the underlying architecture are explored.

Because the search algorithms may try to assess infeasible design points during the DSE process, a feasibility checking module is necessary to analyze the feasibility of design points according to the user constraints files. If infeasible design

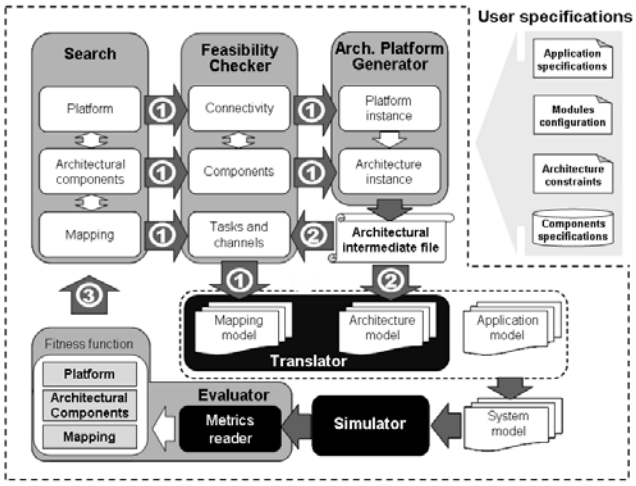

Figure 1. The NASA infrastructure.

points are detected, then NASA's repair mechanisms can be applied to convert them to feasible ones with only a minimum influence on the run-time of the framework.

Subsequently, the Architectural Platform Generator and Translator modules allow for evaluating selected design points using system-level simulation. To this end, the Architectural Platform Generator combines the design decisions on platform and architectural components to compose complete architecture descriptions. Subsequently, the Translator translates this internal representation of design point (i.e., architecture, application and mapping description) into a file-based system model as required by the (external) system-level simulator. So, integrating a new system-level simulator simply requires a new Translator module that generates the simulator-specific configuration file(s) that specify a design point. This is why two kinds of module colors can be identified in Figure 1: simulation-tool-dependent (black) and simulation-tool-independent (gray) modules.

Using the design specifications generated by the Translator modules, the system-level simulator is used to obtain different system metrics, like performance, energy consumption, etc. The simulation results are used by NASA's Evaluator module to evaluate the "fitness" of design points, providing feedback to the Search module and guiding it in a systematic way through the design space. In the next section, we provide more details about the implementation of each of the aforementioned modules.

V. IMPLEMENTATION OF NASA

Before explaining the different modules in detail, the interfaces used in the NASA framework are briefly presented first.

A. Interfaces

The file-based interfaces (represented by the numbered arrows in Figure 1) in NASA, which allow its modules to operate as black boxes, are an essential element to yield a flexible and extensible framework. Three kinds of interfaces are used in NASA: the *architectural intermediate file* (arrows with number 2) is used for communication between the Generator and Translator, the *fitness file* (arrow with number 3) links the Evaluator with the Search module, and the *design-options file* (arrows with number 1) is used in all sub-modules of both the Search module and the Feasibility Checker. In our current implementation, the information in all these files is syntactically described in a proprietary format, although an XML-based format will be used in our future work.

In NASA, both the design-options and fitness files share the same format. They use a dynamically sized, string-based representation format to describe a specific design point. Such a description specifies the platform instance, the architecture instance of that specific platform, and the mapping of the application onto the specific architecture. For a 3-level design space exploration (i.e., exploration at platform, architectural components and mapping levels), the designer can configure the number of the search algorithms to use, i.e., using a single one for all levels (dimensions) or a separate one for each dimension. The default option in NASA is to use one search algorithm per dimension. For this reason, the design-options and fitness files use one descriptive string per design space dimension (i.e., separate strings to describe the platform instance, architecture instance, and mapping). If the designer decides to use less than one search algorithm per dimension, then adapter modules will automatically translate the input and output of the Search module to match the string per dimension format. An example of the descriptive strings is shown in Figure 2, for three (Figure 2a) and one (Figure 2b) search algorithms (SA) in the Search module.

The length of the descriptive string for each dimension may also vary. Using the example shown in Figure 2, it is evident that the length of the string describing the mapping depends on the number of tasks and communication channels in the application. Similarly, the length of the string describing the architecture instance is dependent on the number of processors and memories in the platform.

Finally, the values inside the descriptive strings do not hard-code absolute values but are indirections to table entries (also illustrated in Figure 2a). This means that, for example, in the case of the mapping dimension, the string elements do not directly hard-code the processors (including their exact type) onto which application tasks are mapped. Instead, the string elements point to entries in a processor table. Hence, this allows the designer to e.g. change the types of processors or add a new type without the need to adapt any module implementation. Clearly, this makes the approach more re-usable and extensible.

The last important interface in NASA is the architectural intermediate file. It describes an architecture instance in a single file, which it is gradually constructed using the platform and architectural components strings. Moreover, it is also used to check mapping feasibility. Note that platforms are not fixed entities in NASA but often are also part of the exploration. Therefore, the Feasibility Checker requires, e.g. connectivity information, specifying which and how processing elements are connected, and which memories are shared by which

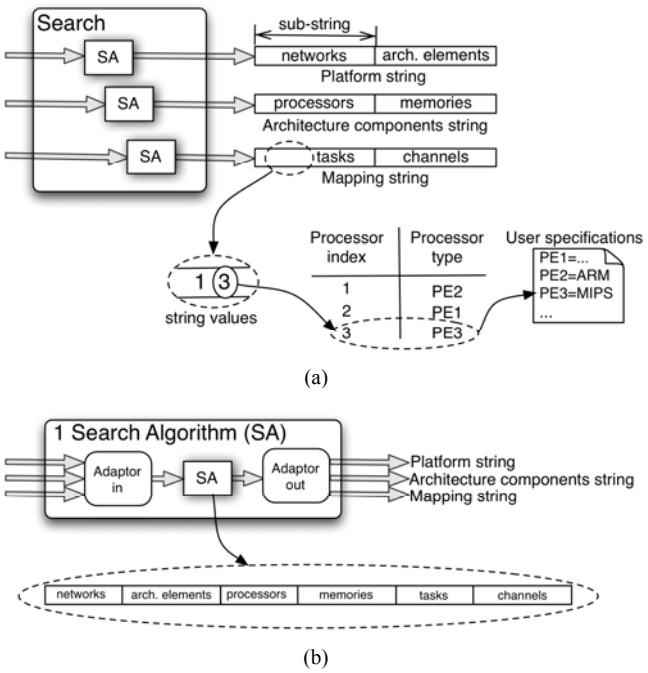

Figure 2. Search Algorithms (SA) and search strings in NASA.

processing elements. This information is needed to detect and repair infeasible mappings, as will be explained in Section 5.3. Finally, the architectural intermediate file and the checked mapping string (which defines the mapping of application tasks and communication channels onto the specified architecture instance) are used by the Translator to generate a simulation-specific description of the design point in question.

B. Search module

This module performs the actual search through the design space, iteratively pinpointing (a set of) design instances that need to be evaluated by means of system-level simulation. As mentioned before, NASA applies a dimension-oriented design space exploration approach. This means that each dimension (platform, architectural components, and mapping) can be co-explored simultaneously using a single search algorithm, or using multiple and possibly different search algorithms for the various dimensions. The designer simply selects and configures the appropriate number and type of search algorithms to be used in the exploration process, according to the characteristics of the design space or each of its dimensions.

In practice, a multitude of search algorithms can be used (via a simple plug-in mechanism) for searching one or more design space dimensions: from exhaustive search or random search, to heuristic search methods. In this paper, we focus on a Search module implementation based on genetic algorithms (GA). GA-based DSE has been widely studied in the domain of system-level design [6,9,17,18], and it has been demonstrated to yield good results.

At this point, we would like to highlight the advantages of NASA's dimension-oriented design space exploration approach. First of all, it provides flexibility as it allows DSE experiments to fix one or more levels of exploration if needed. For example, to find an optimal platform configuration, one could fix the platform level and perform exploration at the architectural components and mapping levels. A second advantage is that one can tailor the search algorithms for the given dimension (level) they explore, according to the characteristics of that dimension. It should be noted, however, that despite of using one search algorithm per dimension, we do *not* perform the system-level design space exploration as multiple independent explorations. Decisions in the Search module should be made by simultaneously taking into account the results from *all* dimensions. This feedback information is provided by the Evaluator module, which will be explained in Section V-G.

C. Feasibility Checker

Independent of the number of search algorithms used in NASA, the Feasibility Checker always receives x sets of design decisions in the form of descriptive strings from the Search module, where x is equal to the number of dimensions of the explored design space ($x=3$ for our platform, architectural components and mapping dimensions). For example, in the case of a single search algorithm is used in the Search module, adapter modules will be automatically plugged in to translate the inputs and outputs of the Search module to comply to the x-strings interface, as is illustrated in Figure 2b.

The main task of the Feasibility Checker is to detect infeasible design points and to repair those design points using heuristic algorithms. During this checking process, all sets of design decisions (strings) are checked in a hierarchical fashion. For example, for our 3-level exploration as shown in Figure 2a, the platform string is checked first to determine whether or not the specified platform template (to be discussed in more detail in the next section) contains a valid topology, and e.g. does not contain isolated islands of components. Next, the architectural components string is checked to determine whether or not the number and types of selected architectural components in the platform template comply with the constraints provided by the user (e.g., if a design point deploys 4 ARM processors while the user has specified that there are only 2 ARM processors available, then we have an infeasible design point). Finally, the mapping string is checked for infeasibility, e.g., when application tasks are mapped onto processing elements that have not been allocated in the platform, or in the case there is no shared memory to map a logical communication channel between two tasks that have been assigned to different processing elements. So, each design point is globally checked, i.e., taking all dimensions of the design point into account.

If an infeasible design point is detected, then different kinds of repair mechanisms can be applied, dependent on the dimension where the problem occurs. Note that different repair techniques can also produce different feasible solutions from the same infeasible design option. In our current implementation, we use heuristic minimum-distance repair techniques, which introduce a minimum number of modifications to an infeasible design point string in order to obtain a feasible one. As a consequence, our repair techniques only have a minimal effect on the run-time of the framework. In the aforementioned infeasible mapping example (i.e., no reachable memory for two communicating tasks), only one of those two application tasks should be relocated if a feasible mapping can be derived from such a repair.

D. Architectural Platform Generator

The architecture description of a design point is created in two steps: platform or topological template generation and architecture instance generation, as illustrated in Figure 3.

The basic building block of these descriptions is the so-called Basic Topology Unit (BTU). A BTU is a logical "pattern" consisting of a network container (the gray component) and a variable number of element containers (the white blocks). These element containers can, in a later stadium, be instantiated as architectural components such as processors and memories. As shown in Figure 3, the number of element containers in a BTU is dependent on the specified user constraints, like the maximum number of processors and memories in a platform.

The BTU is replicated a number of times to form a meta-platform. Here, the number of BTU replications is dependent on the maximum number of networks, as specified by the user. The meta-platform basically is capable of describing all allowed platform instances. The generation of the BTU and the meta-platform is performed statically (but automatically), before the actual DSE process. During the platform exploration, the meta-platform is used to generate topological template instances. To this end, the search algorithm used in platform dimension makes a number of decisions to instantiate a topological template: it sets the number of element and network containers in the platform. Moreover, the network type(s) in the platform is/are also determined, and a type-

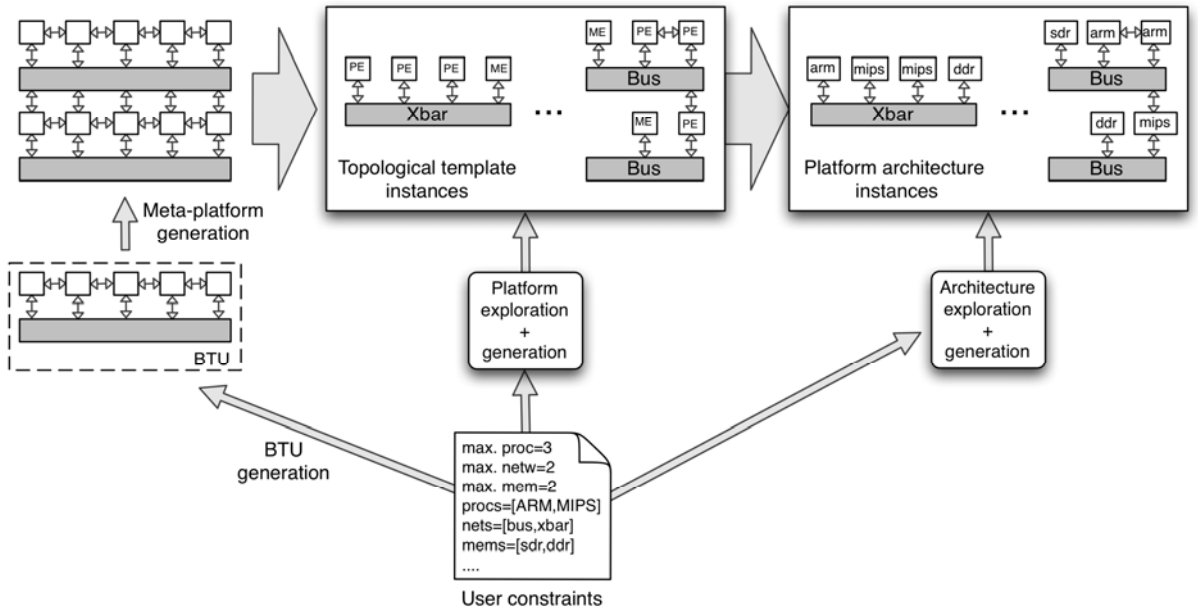

Figure 3. Generating topological templates and architecture instances.

classification of the element containers is made. The latter means that for each allocated element container in the BTUs, it is indicated whether it contains a processor element (PE) or a memory element (ME).

During the architecture exploration, the topological templates are further refined. At this stage, the actual component types of the element containers in a template are added. In Figure 3, this means that, e.g., a processor element allocated in an element container either becomes an ARM or MIPS processor, and the memory elements either SDRAM or DDRAM.

Finally, in order to obtain the complete specification of the design point, the different application tasks and communication channels are bound to the processing and memory elements of the architecture instance in the third step, which is done in the Translator module.

E. Translator

The output obtained from the Architectural Platform Generator module and the feasible mapping strings (checked in the Feasibility Checker module) are used to produce the input files for the system-level simulator that is plugged into the framework. To this end, the Translator module converts NASA's internal format of a design point to a simulator-specific format. This implies that the integration of a new system-level simulator in NASA only requires the adaptation of the Translator module, while all other modules remain unaffected.

F. Simulator

At this moment, we have integrated a SystemC-based system-level simulation environment, called CASSE [3], in NASA. Another system-level simulator, called Sesame [4], is in the process of being integrated. Both simulation tools focus on facilitating efficient system-level DSE of embedded multimedia systems, allowing rapid performance evaluation of different architectural designs, application to architecture mappings, and hardware and software partitioning.

In order to integrate a system-level simulator in NASA, it is required that the simulator allows for explicitly describing the design points that need to be simulated using some kind of file format. For example, for both CASSE and Sesame, three kinds of input files are used: an architecture description file, an application description file, and a mapping file. Obviously, these can be perfectly generated by a customized Translator module, as it was explained in section V-E.

Two key properties could be highlighted for both tools. In first place, the above mentioned specification files are parsed by CASSE and Sesame at runtime to simulate a specific design point. So, changes in the description files do not require any recompilation effort. Evidently, this allows for evaluating design alternatives during the exploration process in a completely automatic way, without any human intervention. Secondly, both tools ensure deadlock-free task mappings and schedules for feasible design points. The interested reader is referred to [2] for an overview of existing system-level simulators, and to [3,4] for more detailed information about CASSE and Sesame.

G. Evaluator

System-level simulators can provide a variety of metrics, such as performance, cost/area, and power/energy consumption. All these metrics, or objectives, can be used in system-level DSE, which then yields a multi-objective optimization problem.

The essence of the Evaluator component is to provide the feedback about the quality of a set of evaluated design points to the Search module, thereby influencing the search decisions taken by the latter during the exploration process.

Separating the Evaluator from the Search component again provides flexibility and enhanced reusability of the components in NASA. It allows for easily changing the optimization objectives or the function that quantifies the quality of a design point – using the various metrics such as performance, power

and cost – without affecting the other components. Such a function is typically referred to as the *fitness function*. The Evaluator also provides the flexibility to, e.g., use a single fitness function for all search algorithms in the Search Module, or to deploy a different, and possibly tailored, fitness function per search algorithm. Clearly, if multiple fitness functions are used, then these should be defined in a coherent way with respect to each other (i.e., avoiding conflicting fitness functions) in order to safeguard convergence.

VI. EXPERIMENTAL RESULTS

A. Experimental setup

To demonstrate the NASA framework and illustrate its distinct aspects, we present a number of DSE experiments in this section. In Table I, the most important user constraints and parameters for these experiments are listed. The studied MP-SoCs may consist of up to 6 processor elements (PE) of the types ARM, PowerPC, or MIPS, up to 3 memory elements (ME) of either single or double data-rate type, and up to 4 networks of three types (bus, fully connected, or a customized network consisting of a bus and point-to-point links). A real-life multimedia application is used to be mapped onto the target MP-SoC. This application is an optimized version of the computer vision algorithm presented in [11]. Basically, this visual tracking algorithm has a real-time requirement (25 frames/sec), and applies a correlation or block matching technique to continuously track a specific target in the incoming image frames. This application consists of 7 tasks and 12 communication channels. The block or pattern size and frames size used in our experiments are 24×24 and 320×240, respectively.

With respect to the search algorithm(s) we use for exploration, we focus on implementations based on genetic algorithms (GAs). We use a proprietary implementation of the GAs, but any existing GA such as SPEA2 or NSGA-II [9] could also have been used. Our DSE experiments were performed using a single GA for the platform, architectural components and mapping dimensions, which is a traditional form of system-level DSE, as well as using a GA per dimension (i.e., 3 GAs in total).

The crossover and mutation operators in our GAs are performed at the granularity of entire sub-strings (see Figure 2) in a string that describes the topological platform, architectural components or mapping. These operators are applied according to their associated probabilities (*pc*: probability of crossover, and *pm*: probability of mutation). Further, the GA can perform either a 1-point or a 2-point crossover, and supports two types of mutation. In "simultaneous" mutation (M=1), a single random position is simultaneously changed in every sub-string. In "independent" mutation (M=6), the mutation probability is used for every of the six sub-strings to determine whether it is mutated or not. In the case of three GAs are used for exploration, different and customized values for the probabilities *pc* and *pm* can be used within each GA.

Moreover, when using three GAs, there also are many ways of linking the individuals of each dimension to form a design point to be simulated. For example, using a pyramidal technique, all (or some) of the individuals in the mapping dimension are linked with each of the individuals in the architectural components dimension, while the latter are again all linked with each of the individuals in the platform dimension. However, this means that the number of design points to be evaluated in each search iteration grows exponentially with the population size of each dimension.

The other extreme is a pure one-to-one individual linking technique. This means that each individual in each dimension is linked to only one individual in the other dimensions. Clearly, this significantly reduces the number of required evaluations. However, this approach may suffer from a possible convergence problem due to under-exploration. For example, let A be a design point formed by platform A^p, architectural components A^c and mapping A^m, while B is another design point formed by platform B^p, architectural components B^c and mapping B^m. If it turns out after a single simulation that the fitness value of A is better than that of B, then it clearly does not mean that platform A^p or architectural components A^c are always a better choice than B^p and B^c. Thus, to avoid such an under-exploration problem, more evaluation data should be collected for the platform and architecture dimensions before discarding any of their individuals.

To address the under-exploration problem in hierarchical design space explorations with multiple search algorithms, we use a variant of one-to-one individual linking technique. In this technique, an individual from each dimension is again linked to a single individual in the other dimensions. But unlike the pure one-to-one technique, only the individuals from the dimension of the lowest abstraction level (i.e., the mapping dimension in our case) are evaluated *and updated* during every search

TABLE I. PARAMETER SETTINGS IN OUR EXPERIMENTS.

Parameter	Nr.	Types	Values
PE	≤ 6	3	ARM, PPC, MIPS
ME	≤ 3	2	DDR, SDR
Networks	≤ 4	3	Bus, Fully-connected, Customized-network
App. Tasks	7	-	-
App. Channels	12	-	-
Dimensions *(β)*	3	-	Platform, architectural components and mapping
Search algs. *(SA)*	1 or 3	1	Genetic algorithms
GA Selection *(S)*	1	1	Proportional with elitism
GA Crossover *(C)*	1	2	1-point and 2-point
C probability *(pc)*	5	-	[0.1,0.3,0.5,0.8,1.0]
GA Mutation *(M)*	1	2	Simultaneous (M=1) and Independent (M=6)
M probability *(pm)*	5	-	[0.1,0.3,0.5,0.8,1.0]
Collecting iterations (δ_{arc})	1	-	2, architectural components dimension
Collecting iterations (δ_{pla})	**1**	-	4, platform dimension
Search iterations *(I)*	41	-	-
Population size	10	-	Nr. of individuals per iteration
Simulation tool	1	-	CASSE

iteration. The search algorithms for the higher-level dimensions (i.e., the platform and architectural components dimensions) keep collecting the fitness values of their individuals (for different mappings) without actually changing their individuals during a specified number of iterations, referred to as the *collecting iterations* (δ). Only when the search has reached δ iterations, the individuals are updated, after which the process starts again. Obviously, as explained in section V-B, the higher the abstraction level, the more design alternatives can be derived for a single design point (e.g., a multitude of architecture instances can be obtained from a single platform), and consequently, the higher the value of δ should be. From the above we can conclude that there exists a tight connection between the different search algorithms and their respective fitness functions. Formally, these relations can be defined as follows:

$$y_{L_i} = f_L(x_1, x_2, ..., x_k); \quad \forall i = 1..I$$

$$\begin{cases} y_{j_i} = f_j(x_1, x_2, ..., x_k) = \sum_{q=1}^{\delta_j} y_{L_q}; & \forall i = 1, \delta_j, 2\delta_j..I \quad and \quad \forall j \neq L \\ \delta_z > \delta_w; & \forall z, w = 1..\beta \quad and \quad z \supset w \end{cases}$$

where y_{L_i} is the fitness value of an individual of the lowest-level dimension (the mapping dimension in our case) in the iteration i, I is the total number of search iterations, x_k represents the value of the metric k used in the fitness function f, y_{j_i} is the fitness value of an individual in any dimension other than the lowest one, and δ_j represents the collecting iterations for the individuals of dimension j. Moreover, for a given range of dimensions β, the number of the iterations needed for collecting fitness information for dimension z (e.g., platform) should be bigger than the number of iterations needed for dimension w (e.g., architecture) if z has a higher abstraction level than w.

If all the GA parameters in Table I are taken into account, a large number of experimental combinations can be performed. Due to space limitations, however, we can only present a selection of four NASA configurations in this paper. The nomenclature used to denote these configurations is "SAgaCxM", where the meaning of each capital letter is defined in Table I. For example, "3ga1x6" refers to the configuration with 3 GAs that simultaneously explore the platform, architectural components and mapping dimensions, a 1-point crossover, and an "independent" mutation (M=6).

B. Results

In a first experiment, all possible combinations of the *pc* and *pm* values, as listed in Table I, have been evaluated. This results in 25 experiments for each of the four mentioned NASA configurations, where a maximum of 410 simulations (41 iterations x 10 individuals per iteration) have been performed for each experiment. The CASSE simulator – which dominates the execution time of our DSE experiments – requires on average 40 seconds to simulate a single design point on a PC, with a Pentium IV processor at 1,6 GHz and 2 GB main memory, running Linux. Moreover, each experiment has been executed twenty times using different sets of initial populations. To simplify the graphic representation of the

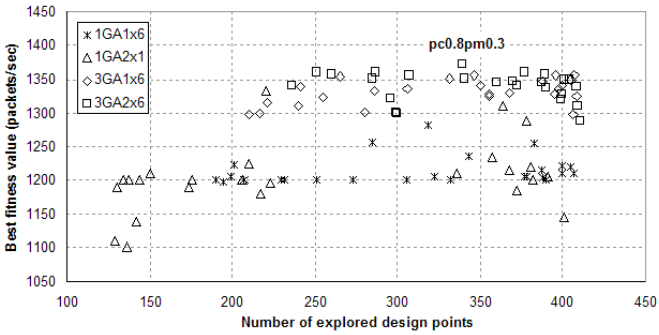

Figure 4. DSE results for six NASA configurations.

results and the explanation of the examples in this section, the fitness value in our experiments only takes a single system metric into account, namely performance. We would like to stress, however, that multi-objective optimization can also be perfectly addressed with NASA.

The results of the above experiment are shown in Figure 4. It shows a scatter-plot with the average number of *different* explored design points on the x-axis and the average of the best fitness values (in terms of processed data packets/sec) on the y-axis for each of the above mentioned experiments. If a minimum of 1250 packets/sec has to be processed to satisfy the real-time requirements (25 frames/sec) of the studied application, then using a 3 GA-based searching approach in NASA not only provides the design alternatives with the best fitness values but the diversity of the explored design points is also largest. Exploring the same design space with a traditional, single GA approach, optimal and near-to-optimal architectures are less often found. This is mainly due to a smaller diversity of explored design points.

Figure 5 zooms in on a particular set of experiments, where all configurations use the values *pc*=0.8 and *pm*=0.3. With these probabilities, the "3ga2x6" configuration finds the design point with the best overall fitness value (see Figure 4). The bars in Figure 5 represent the average accumulated number of different design points explored (i.e., the aggregated diversity of the population), while the lines show the average of the best fitness values obtained in each iteration. Investigating the fitness values, it can be seen that the 3 GA-based experiments gradually but progressively reach the best fitness values. In the 1 GA-based experiments, on the other hand, mostly design

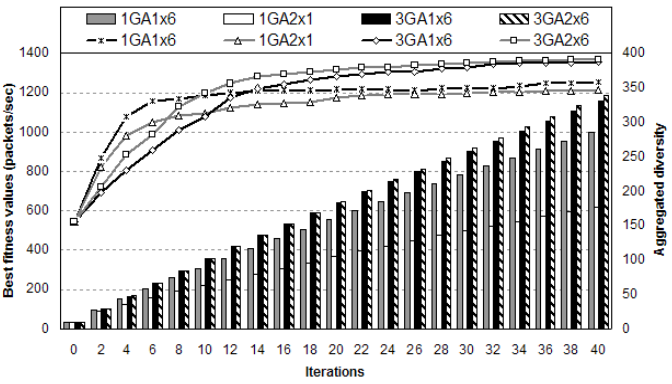

Figure 5. Best fitness values and diversity per iteration.

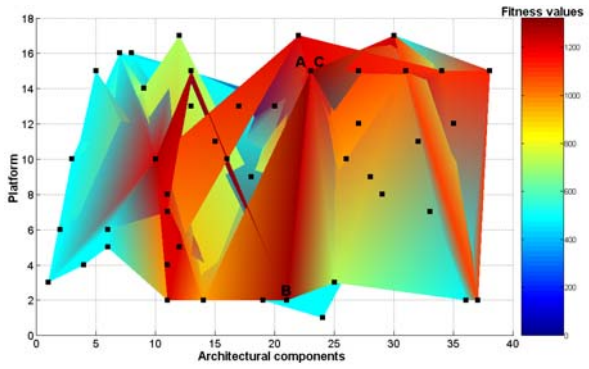

Figure 6. Explored design space in 2D view (architectural components vs. platform).

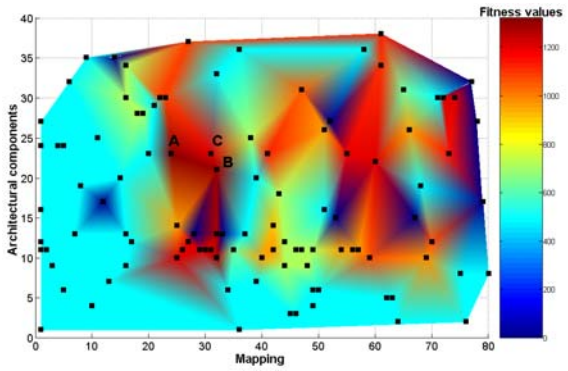

Figure 7. Explored design space in 2D view (mapping vs. architectural components).

points of a lower fitness are reached, and they often do not improve anymore after the twentieth iteration. The latter could indicate that the GA is trapped in a local optimum. In that case, the lower diversity of the populations in the 1 GA-based experiments prevents the search algorithms to escape from such local optima. In Figure 5, it can also clearly be seen that the 1 GA-based experiments, and especially those with "simultaneous" mutation (M=1), have a much lower aggregated diversity than the 3 GA-based experiments. At this point, we also would like to mention that although the size of the design space (approximately 10^{12} alternatives, as mentioned in Section III) is much bigger than the number of individuals actually evaluated (maximum 410) in these experiments, the results in Figure 5 clearly indicate that the GA-based DSE in our experiments converges to a (local or global) optimum after only a relatively small number of iterations.

A proper population diversity not only helps in avoiding the above mentioned local optimum problem, but it also allows designers to more easily compare the architectural characteristics of the evaluated design points. That is, it can be very useful for a designer to distinguish the architectural similarities of the design alternatives with good fitness values. Obviously, at this point, the higher the population diversity during the exploration, the more information can be extracted from the explored design space.

To illustrate the above, please consider Figures 6 and 7. Note that the typical NASA output after each experiment is a set of simulated design points, which form a surface that approximates the landscape of the design space. Since our experiments are based on 3D exploration, we present two 2D views of the resulting surface. Figure 6 shows the fitness values when only considering the architectural components dimension and platform dimension of design points. The *x* and *y* axes of Figure 6 contain the explored instance numbers of the architectural components and platform dimensions. So, for example, 17 different platform instances have been explored. The fitness value is color coded, ranging from red (high fitness value) to blue (low fitness value). In a similar fashion, Figure 7 shows the fitness values for the mapping dimension and architectural components dimension of design points.

In Figures 6 and 7, the dark red areas clearly pinpoint the design points with the higher fitness values. For example, three of these design points (A, B, C) have been marked in both Figures 6 and 7, and their respective architectures and mappings are shown in Figure 8. From Figure 8, it can immediately be seen that design points A and C share the same underlying platform architecture, but the application mapping of A (achieving 1321 packets/sec) is more efficient than the mapping of C (achieving 1202 packets/sec). Moreover, it can be seen that design points A and C are over-dimensioned in the sense that not all resources are actually used by the application. This is also illustrated by the very similar performance of design point B (1289 packets/sec), which only uses a subset of the resources in design points A and C. Evidently, design point B can also be further optimized, both in terms of platform architecture (it is still slightly over-dimensioned) and in terms of mapping. Such refined optimization can be performed in a next, more detailed phase of exploration experiments where, e.g., the platform is fixed and/or additional objectives (such as cost) are taken into account. Note, however, that these

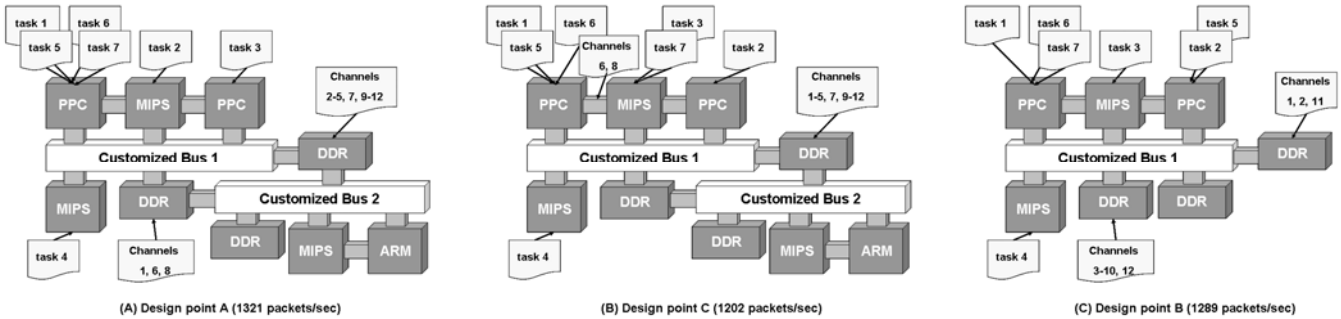

Figure 8. Examples of design points found by 3ga2x6 DSE with pc=0.8, pm=0.3.

experiments focus on demonstrating NASA's DSE process with various processing and network element types having different computational and communication characteristics, rather than performing a real DSE experiment for a certain target system. Moreover, although not studied in our experiments, we also note that NASA as well as the incorporated simulators (i.e., CASSE and Sesame) support dedicated hardware components as processing elements in DSE experiments.

To summarize, even when exploring a relatively small number of design points, NASA's output (Figures 6 and 7) can provide a good insight of where the sweet spots in the design space are, which can then be further explored in more detail by, e.g., fixing one or more search dimensions in the exploration. Finally, we would like to stress that all these experiments have been performed in a fully automatic fashion, only providing parameter settings and constraints such as shown in Table I.

VII. CONCLUSIONS

In this paper, we addressed the lack of a generic, flexible, and re-usable infrastructure to facilitate and support system-level MP-SoC design space exploration (DSE) experiments. To this end, we have presented a system-level MP-SoC DSE support infrastructure, called NASA. This highly modular framework uses well-defined interfaces to easily integrate different system-level simulation tools as well as different combinations of search strategies in a simple plug-and-play fashion. Moreover, we described NASA's dimension-oriented DSE approach, allowing designers to configure the appropriate number of, possibly different and tailored, search algorithms to simultaneously co-explore the various design space dimensions. The result is a flexible and re-usable framework for the systematic exploration of the multi-dimensional MP-SoC design space, starting from just a set of relatively simple user specifications. We have also demonstrated several distinct aspects of NASA by presenting a number of DSE experiments in which we, e.g., compared NASA configurations using a single search algorithm for all design space dimensions to configurations using a separate search algorithm per dimension. These experiments have shown that the latter multi-dimensional co-exploration can find better design points and evaluates a higher diversity of design alternatives as compared to the more traditional approach of using a single search algorithm for all dimensions. As future work, we plan to integrate other types of search algorithms into NASA, e.g., through the integration of the PISA optimization framework [18], as well as to perform additional deployment case studies of NASA.

REFERENCES

[1] G. Martin, "Overview of the MPSoC Design Challenge", *Proc. of Design Automation Conference (DAC'06)*, Jul. 2006.

[2] M. Gries, "Methods for evaluating and covering the design space during early design development", *Integration, the VLSI Journal*, Vol. 38, no. 2, pp. 131-183, Dec. 2004.

[3] V. Reyes, T. Bautista, G. Marrero, P. P. Carballo and W. Kruijtzer, "CASSE: A System-Level Modeling and Design-Space Exploration Tool for Multiprocessor Systems-on-Chip", *Euromicro Symposium on Digital System Design (DSD'04)*, pp. 476-483, 2004.

[4] C. Erbas, A.D. Pimentel, M. Thompson and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures", *EURASIP Journal on Embedded Systems,* no. 1, pp. 2-2, Jan. 2007.

[5] C. Lee, S. Kim and S. Ha, "A Systematic Design Space Exploration of MPSoC Based on Synchronous Data Flow Specification", *Journal of Signal Processing System*, Vol. 58, no. 2, pp. 193-213, Feb. 2010.

[6] M. Palesi and T. Givargis, "Multi-objective Design Space Exploration Using Genetic Algorithms", *Proc. of the international symposium on Hardware/software codesign (CODES'02)*, pp. 67-72, May. 2002.

[7] K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 19, no. 12, pp. 1523-1543, Dec. 2000.

[8] B. Kienhuis, E. Deprettere, K. Vissers and P. van der Wolf, "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures", *Proc. of the IEEE Int. Conference on Application-Specific Systems, Architectures and Processors*, pp.338, Jul. 1997.

[9] C. Erbas, S. Cerav-Erbas and A.D. Pimentel, "Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design", in *IEEE Transactions on Evolutionary Computation*, pp. 358-374, Vol. 10, no. 3, Jun. 2006.

[10] Z. J. Jia, T. Bautista and A. Nuñez, "Real-Time Application to Multiprocessor-System-on-Chip Mapping Strategy for a System-Level Design Tool", *IEE Electronic Letters,* Vol. 45, no. 12, pp. 613-615, 2009.

[11] Z. J. Jia, T. Bautista, A. Nuñez, C. Guerra and M. Hernandez, "Design space exploration and performance analysis for the modular design of CVS in a heterogeneous MPSoC", *Proc. of the Conference on Reconfigurable Computing and FPGA (ReConFig 2008),* pp. 193-198, Dec. 2008.

[12] S. Mohanty, V. K. Prasanna, S. Neema and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation", *Proc. of Languages, compilers and tools for embedded systems: software and compilers for embedded systems (LCTES'02-SCOPES'02)*, Jun. 2002.

[13] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri and L. Benini, "An integrated open framework for heterogeneous MPSoC design space exploration", *Proc. of the Design, Automation and Test in Europe (DATE'06),* pp. 1145-1150, Mar. 2006.

[14] www.multicube.eu

[15] L. Thiele, I. Bacivarov, W. Haid and K. Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems", *Proc. 7th Intl Conference on Application of Concurrency to System Design (ACSD 2007)*, pp. 29-40, Jul. 2007.

[16] G. Palermo, C. Silvano and V. Zaccaria, "A Flexible Framework for Fast Multi-objective Design Space Exploration of Embedded Systems", *PATMOS 2003,* Vol. 2799, pp. 249-258, Sep. 2003.

[17] J. Madsen, T.K. Stidsen, P. Kjarulf and S. Mahadevan, "Multi-Objective Design Space Exploration of Embedded System Platforms", *IFIP*, Vol. 225, pp. 185-194, 2006.

[18] S. Künzli, L. Thiele and E. Zitzler, "A Modular Design Space Exploration Framework for Embedded Systems", *IEE Proc. Computers & Digital Techniques*, pp. 183-192, 2005.