Overview of the MPSoC Design Challenge

Grant Martin
Tensilica, Inc.
3255-6 Scott Boulevard
Santa Clara, CA 95054-3013 USA
+1-408-327-7323
gmartin@tensilica.com

ABSTRACT

We review the design challenges faced by MPSoC designers at all levels. Starting at the application level, there is a need for programming models and communications APIs that allow applications to be easily re-configured for many different possible architectures without tedious rewriting, while at the same time ensuring efficient production code. Synchronisation and control of task scheduling may be provided by RTOS's or other scheduling methods, and the choice of programming and threading models, whether symmetric or asymmetric, has a heavy influence on how best to control task or thread execution. Debugging MP systems for the typical application developer becomes a much more complex job, when compared to traditional single-processor debug, or the debug of simple MP systems that are only very loosely coupled. The interaction between the system, applications and software views, and processor configuration and extension, adds a new dimension to the problem space. Zeroing in on the optimal solution for a particular MPSoC design demands a multi-disciplinary approach. After reviewing the design challenges, we end by focusing on the requirements for design tools that may ameliorate many of these issues, and illustrate some of the possible solutions, based on experiments.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application-Based Systems**]: Real-time and embedded systems

C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: SIMD, MIMD

General Terms

Measurement, Performance, Design, Experimentation, Languages, Verification.

Keywords

MPSoC, Multi-Processor System-on-Chip, System-Level Design, SLD, Electronic System Level design, ESL, MP-debug, design space exploration

DAC 2006, July 24-28, 2006, San Francisco, California, USA.

1. INTRODUCTION

Increasingly, the design of embedded systems and System-on-Chip devices (SoC) is based on utilising multiple processors. What has been dubbed "MPSoC" [1] is becoming a much more prevalent design style, to achieve tight time-to-market design goals, to maximise design reuse, to simply the verification process and to provide flexibility and programmability for post-fabrication reuse of complex platforms. Sometimes these processors may be fixed Instruction Set Architecture (ISA) processors; sometimes they may be configurable, extensible processors [2,3]. Very often there is a mix of processor types. The now classical RISC+DSP combination used in baseband applications in cellphones is an early example of MPSoC, and a good illustration of the evolution of such devices.

Modern cell phones may have four to eight processors, including one or more RISCs for user interfaces, protocol stack processing and other control functions; a DSP for voice encoding and decoding and radio interface; an audio processor for music playback; a picture processor for camera functions; and even a video processor for new video-on-phone capabilities. In addition, there may be other deeply embedded processors substituting for other functions traditionally designed as dedicated hardware blocks. Extensible processors in particular are proving to be flexible substitutes for hardware blocks, achieving acceptable performance and power consumption. Thus these devices are a good illustration of heterogeneous MPSoC, and their demanding requirements for low cost, reasonable performance, and minimal energy consumption illustrates the advantages of using highly application-specific processors for various functions.

This shift to an increasingly processor-, and multi-processor-, centric design style, poses many challenges for system architects, software and hardware designers, verification specialists and system integrators. These may best be met by revisions to old tools and methods to deal with MPSoC complexities; by new tools and methods, working at the same abstraction levels; and by moving up in abstraction to take advantage of new design approaches.

2. DESIGN CHALLENGES FOR MPSOC

2.1 Programming Models

Since MPSoC design is by its nature processor-centric, and thus software-centric, the first, and by all accounts, most difficult design challenge for these devices lies in the programming model(s) that are required to map application software into effective implementations.

What makes MPSoC difficult to program? Two factors are key: concurrency, and "fear of concurrency" [4, 5]. Software developers have been well-trained by sixty years of computing history to think in terms of sequentially defined applications code, with ever-faster computers on which to run it. Early embedded systems with a single processor continued this heritage. It's very easy to think sequentially; the fundamental nature of an algorithm is to describe a sequence of steps to solve a problem. Most programming languages encourage sequential thinking.

Contrast the sequential nature of classical programmes with the concurrent possibilities opened up by MPSoC. Here we can distinguish two major classes of MP systems: "symmetric multi-processing" (SMP) in which multiple processors or processing cores share a common view of main system memory, and multiple processes or threads execute within global, shared coherent memory; and "asymmetric multi-processing" (AMP), in which processors are usually much more loosely coupled, may have quite different ISAs, and usually have dedicated local memory resources. But whatever the MP model, multiple tasks are executing on different processors concurrently, communicating with each other and possibly with central resources asynchronously via a number of mechanisms. At any point in time, clashes of priorities, deadlocking, data starvation, races, and data incoherency may occur. The fictional concurrency offered by conventional operating systems or RTOS's running on a single processor, where the transfer of control from one executing process to another is handled in an orderly fashion, is replaced by a raging sea of simultaneously executing tasks, any one of which might interact with another in a most unpredictable fashion. Decomposing an application described in a serial fashion into a set of concurrent or parallel tasks that can co-operate in an orderly and predictable way is one of the most difficult jobs imaginable, and despite forty or more years of intensive research in this area, there are very few applications for which this can be done automatically.

Programming models are chosen precisely to bring some of the variability involved in concurrency under control. In the SMP model, processors are usually identical or at least share a common ISA, and thus threads can be assigned and re-assigned to different processors depending on loading conditions and system-level optimisation criteria such as reducing energy consumption. The common coherent memory model means that this can be done relatively easily; but it is still difficult to ensure that processing is done in the right order, that all dependencies are met without deadlock or starvation, that the tasks are well-balanced and that processor utilisation is reasonable. In the AMP model, processors are often tuned to specific tasks either in a coarse-grained fashion (e.g. DSP vs. RISC for numerically-intensive tasks vs. control-dominated tasks), or in a fine-grained manner (as with configurable and extensible processors where instructions may be added to a core for a very specific set of tasks). These communicate with a variety of mechanisms – shared memory, direct hardware fifos, or specialized signaling – but the same issues of processor balancing, correct ordering of computations, and avoiding deadlock and starvation, still occur. Often programming models are supported by standard Application Programming Interface (API) libraries, such as OpenMP or the Message Passing Interface (MPI), or standardised threading models such as POSIX threads, in order to make applications a little easier to map to different MPSoC architectures, or to port from one architecture to another.

However, it is often the case that the mix of applications to be run on a complex MPSoC device may require a mix of programming models and "models of computation", and choosing the correct one(s) becomes a further issue of design. The correct specification, implementation, decomposition and mapping of applications to MPSoC architectures remains an intensely manual task fraught with peril for the unwary.

2.2 Synchronisation and Control

We have alluded to some of the issues involved in synchronising and controlling multiple concurrent tasks on multiple processor cores in the previous section. Even assuming a good decomposition and mapping of the target applications into multiple tasks or threads running on the target MPSoC architecture, there are higher level issues of synchronisation and control that are important. Will the control be delegated to a Real-Time Operating System (RTOS) or based on an ad-hoc scheduling mechanism? Will the system be built by composing multiple processor-based subsystems drawn from different domains (for example, a video, an audio, and perhaps a communications subsystem) that synchronise via infrequent high level messaging only, or must the tasks work in tight lockstep that requires a high amount of inter-process communication? All the classical issues of RTOS priority management, including deadlock and priority inversion, come into play when the number of tasks exceeds the number of processors, or tasks contend simultaneously for a common pool of critical resources.

To save energy, especially critical for portable embedded systems, it is very desirable to shut down portions of the system when not in use, or scale back processor voltage and frequency to match the processing requirements precisely, via techniques such as Dynamic Voltage and Frequency Scaling (DVFS). This requires taking a holistic view of the overall system and its applications, such that task requirements, processor loading, run rate, and energy usage are precisely balanced. Many techniques have been developed in research and industry for such control, but their application still seems rather ad-hoc.

One key lesson learned from earlier attempts at concurrent systems is the desirability of building systems up from composable subsystems [4], which greatly eases the issues involved in MPSoC design.

2.3 Debugging

Even after we've built an application and mapped it to an MPSoC architecture, we will spend some considerable time debugging it. Here all the issues we have tried to guard against in design come bubbling back to the surface [6]. Despite our best efforts to avoid mismatches in communication, deadlock, process starvation, race conditions, and false sharing issues, inevitably when the application is running on the real system (which may be the real MPSoC, or a cycle-accurate or fast functional model of such a device), something 'bad' will happen. At this point, an MP-capable debugger is both

incredibly useful and often the sole recourse. The requirements for MP debugging are rather complex. Perhaps one of the most subtle points, as discussed in [6], is the need to support "multi-paradigm debugging", or what might be called in other contexts the need to debug across multiple models of computation. Given a composition of loosely coupled subsystems running different portions of an application, communicating with each other in a variety of ways, using a number of different ISA processors and running tasks written in multiple programming languages using multiple programming models, the task of understanding what is going on, and skipping nimbly from paradigm to paradigm while retaining a coherent view of system state in order to track down the root cause of a problem, poses a formidable challenge to concurrent debugging tools.

Of course, all the standard debug tools for starting up, setting breakpoints and break conditions, multiple-processor compositional breakpoints, observation points, tracing, visualising system, application and processor state, and detecting bottlenecks, races, and deadlocking, are required.

2.4 Interactions: System, Applications Software, and Processor Configuration and Extension

Let's take all these MPSoC design challenges and stir them up even further with the opportunity provided by designing Application-Specific Instruction-set Processors (ASIPs). These have emerged from academia [7,8] and the IP industry [2,3], and are also supported by commercial ESL tools such as CoWare (Lisatek), Synfora, Poseidon, and Critical Blue.

When you have the opportunity to configure and extend processor ISAs to better match the performance and power consumption requirements of an application, or portions of an application, and you add to this the ability to have just about as many processors as you want, you have exploded the design space that should be explored in developing an optimal architecture for a particular MPSoC. Suddenly, many more solutions are possible: different decompositions, mappings, communications schemes, and the particular processor configurations, all iteract in ways that make it difficult to decide in an *a priori* fashion which part of the solution space is likely to contain the optimal architecture.

Systematic design space exploration is not something that many system architects have either needed to do in the past, or are familiar with (with exceptions). Very often architectures have been based on "gut feelings" and "the last application", designed with Excel and whiteboards, or are simple derivatives of previous SoC architectures. The rising cost of SoC design and the complexities of MPSoC optimisation make these *ad hoc* approaches much riskier than in the past.

Design methods, and tools to help in this exploration may come from a variety of sources. These include the commercial EDA tools industry (the branch known as "Electronic System Level", or ESL tools); the IP industry (especially the provisioners of processor IP); the embedded software (ESW) tools industry; or indeed, from startups, and from within the engineering groups of systems and semiconductor houses.

3. KEY MPSOC ARCHITECTURAL QUESTIONS

[9] discusses some of the key architectural questions involved in designing and programming an MPSoC system. These can be classified into several categories:

- The number and configuration(s) of processors required for the application. How homogeneous should the architecture be, vs. how heterogeneous.
- Interprocessor communications – choosing the right mix of standard buses, point to point communications, shared memory, and emerging network on chip approaches.
- Concurrency, synchronisation, control and programming model(s). Often multiple models will be appropriate.
- Memory hierarchy, types, amounts, and access methods, along with estimating required latency.
- Special operating modes and controls for power reduction and low energy consumption.
- Application partitioning, use of appropriate APIs and communications models, and associated design space exploration.
- Design and platform scalability. As technology evolves, will the architecture move from 10 to 100 to 1000 or more processors? How often must the application undergo major re-architecting?

Although the commercial ESL, EDA and ESW tools industries may provide some of the generic tool capabilities required to support design space exploration, application analysis and debugging for MPSoC platforms, it is also very likely that the commercial IP industry will be required to offer large components of the solution. This is especially true of highly configurable IP such as extensible processors, where the nature of the IP on offer has a high degree of interaction with the solution required.

4. REQUIREMENTS FOR MPSOC DESIGN SOLUTIONS

4.1 Integrated Development Environment

Most ESW and IP providers, and many ESL tool vendors, are offering integrated development environments (IDEs) to serve as a standard 'cockpit' for software development, or mixed software-hardware-system development. The Eclipse project [10], which started as an open-source IDE for Java, but has added a C/C++ development toolkit (CDT), has grown in popularity as a base for several ESW, IP and ESL tools.

Eclipse was created to be extended and configured for specific uses, design flows and methodologies. It supports a wide variety of basic design views and perspectives and can be complemented by specific views, perspectives, plug-ins and tools. Eclipse is only one example of an IDE that could be used; others are quite common in the ESW tools industry. Whatever the IDE, there are several capabilities important to MPSOC design:

- For configurable and extensible processors, a user interface for processor configuration and extension. Instruction extensions may require specialised compilers along with feedback on

results and integration of the resulting hardware views into the final configuration.

- General software capture, editing, targeting, building and modification capabilities for application software tasks, middleware and project libraries. It must be possible to target tasks to specific processor implementations or instantiations.
- System structural editing for the MPSoC architecture. This includes instantiating processor configurations, memories, communications interfaces, HW fifos, buses, bus interfaces, and a variety of dedicated HW processing blocks and peripherals.
- Simulation control. Generating and running system level simulation models whether on a single processor or the complete MPSOC. This also includes static and dynamic processor profiling and the post-simulation visualisation and analysis of these results. It also should include system-level transaction tracing for bus-based communications transactions as well as more specialised transactions, and an ability to post-process these traces both statistically, generating system level profiling data for performance analysis, and visually for debugging and easy identification of performance problems.
- Advanced MP debugging capabilities, including those discussed earlier, with provisions for setting watch points, trace points and breakpoints on individual processors, software tasks, and other devices; to move easily between simulation and software source; to track the interaction between source code, breakpoints and the simulation, to set up and trip on complex conditions, etc.
- Export capabilities, including export of structural and logical information to 3rd party ESL and ESW tools, and the generation of simulation models, as well as SW export.

4.2 System Structure and Model Generation

Recently there has been an increase of interest in the development and use of standard formats for system structure and IP configuration parameters – what has been called the IP and MPSoC "meta-data". XML-based formats such as SPIRIT [11], derived originally from the XML format used by Mentor Platform Express, have been developed and promoted, although actual industrial usage remains rather low. Although XML tends to be verbose and inelegant, XML-based formats and schemas can be quickly extended, parsed and generated and are an interesting way both to store system structure and parameters and to pass this information between tools.

Another important capability is to be able to generate simulation models, in order to support design space exploration and system level verification and performance analysis at a reasonably high level of abstraction. System-level simulation models for MPSoC will of course utilize Instruction Set Simulation (ISS) models. SystemC has become the *lingua franca* for system level modeling and is increasingly used as the basis for integrating interoperable models into a system level model. The idea of transaction-level modelling [12], although not yet fully standardised by the Open SystemC Initiative (OSCI) or IEEE 1666, is a vehicle for building reasonably fast cycle-accurate system level models, and can be abstracted to offer fast functional models that may be up to 100 to 1000 times faster in performance.

These system-level simulation models are important for simulating the many operating scenarios of a system and its applications, and for tracing and analysing the operating conditions. Fast functional simulation models, sometimes also called 'virtual system prototypes' are particularly desirable for software development and validation.

4.3 MP Programming Models

To allow efficient design space exploration (DSE) of various MP architectures for a particular application, developers may find it useful to have access to abstract programming models that allow the various software tasks to be mapped to processors, scheduled, and to inter-communicate without constantly modifying the source code. Although there are a number of such models and API libraries, there are no well-accepted universal standards that have been adopted in the embedded systems domain. Pipelined dataflow models are one attractive and reasonably simple model that have been studied for years and interesting communications API models such as Philips TTL [13] have begun to emerge. In this model, a limited number of different abstract channels can be supported with varying semantics depending on use models. These are especially useful for AMP applications and platforms. Simultaneous multi-threading (SMT) approaches are also attracting interest, especially for homogeneous SMP clusters of processors with hardware support for thread context-switching and scheduling. It is easy to conceive of platforms with both AMP and SMP characteristics and thus use a heterogeneous set of programming models and abstractions [14].

Of course, an MP-candidate architecture becomes much more interesting if the processors within it support unconventional communications mechanisms such as direct connect queues and ports. It is possible to begin to experiment with direct inference of communications implementation choices for unmapped communications abstractions used in tasks. In addition, mapping abstract communications channel APIs to different possible implementations (for example, a FIFO channel can be mapped to a hardware queue, a shared memory, or some kind of bus-based device) allows flexible design space exploration of a number of different implementation alternatives.

5. EXPERIMENTAL MPSOC SYSTEM-LEVEL SOLUTIONS

These concepts have been implemented in an experimental processor-IP centric design methodology and toolset, specifically oriented towards configurable and extensible processors. This is controlled via an Eclipse-based integrated development environment (Figure 1).

Figure 1 illustrates a table-driven user interface for capturing system structure. Although some tools provide graphical diagrammatic ways of capturing system structure, and this may be a desirable capability in the long term, in the short term it is reasonable at the system level of abstraction to capture MP system structure in a tabular way. Processors and other components, when modeled at the transaction level, have a reasonable and controlled number of high level interfaces, and stitching them up by choosing links in a table is sufficiently easy for MP systems that range up to a few tens of components. In addition, the support of hierarchical subsystem structure with continued use of high level interfaces allows both tabular and graphical system structure editing to be feasible as MP systems grow in complexity.

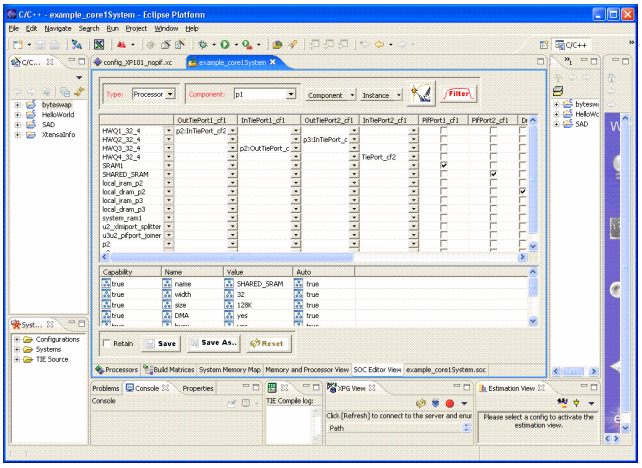

Figure 1: Capturing System Structure

From the system structure captured in the IDE, and from models for configurable processors created from their actual configurations, plus models for other system level components such as memories, routers, queues, arbiters and other devices, it is possible to automatically generate system simulation models of the two kinds mentioned earlier (Figure 2). The first is a SystemC-based cycle accurate system model with extensive tracing capabilities. With this model, the subsystem modeled using configurable processors can be linked to other SystemC models for other portions of the embedded SoC, as long as compatible transaction level models are used, or appropriate wrappers or adaptors between the different notions of 'transaction' are created. Being cycle-accurate, but still using transactions, such a model runs at least 100X the speed of an equivalent RTL simulation. The tracing facilities allow both system level transaction performance to be monitored on a statistical basis, to derive figures on overall system throughput and latencies, and for detailed transaction level debug to take place using a visual depiction of the traces.

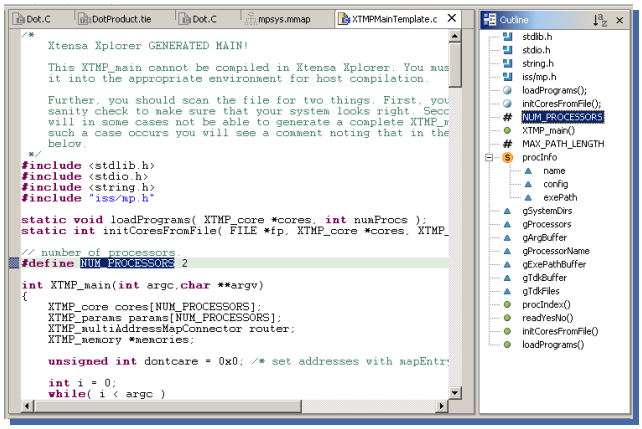

Figure 2: Generated System Simulation Model

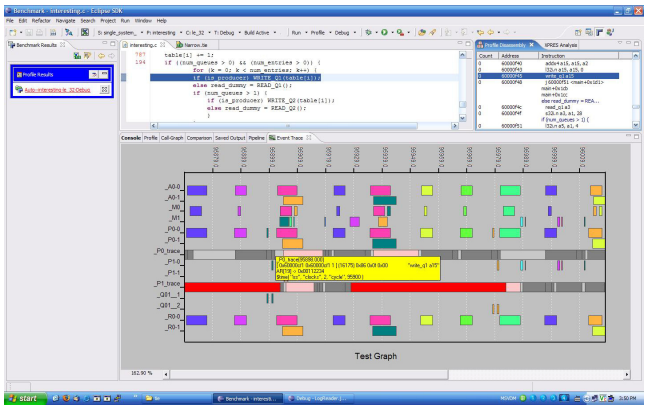

Figure 3: Visualisation of an Event Trace

Alternatively, a fast functional simulation model, which is instruction accurate rather than cycle-accurate, can be generated. This will runs 10-100X faster than a full cycle-accurate system simulation, for a multi-processor system. Such a model is particularly useful for software developers, as long as careful attention is paid to the speed-accuracy tradeoff, and as long as appropriate synchronization models are used. For example, rather than using a fifo queue of fixed depth as in the cycle-accurate simulation, which may stall processor execution in a mis-matched system for many cycles (because it is full when a processor wishes to push more data to it, or empty when a processor wishes to pop data from it), it may be appropriate in this case to use an effectively infinite depth buffer rather than a fixed depth queue. Such a buffer can be called using direct method calls from the fast simulation rather than be an explicitly modeled device. This will be functionally accurate in normal operation, and thus allow software development and verification to proceed.

Figure 3 illustrates a trace file generated in the course of cycle-accurate system simulation. This can be used to monitor and debug system level transactions and to determine the systemic cause for system performance problems. Transaction requests can be examined as they ripple through a hierarchy of devices and their responses can be analysed. Stalls, contention and unusually long delays in transaction responses can be displayed visually as exception conditions. The system level design capability has implemented some of the abstract communications mechanisms discussed previously and is able to map FIFO channels into a variety of implementations including direct hardware queues and shared memories with various locking mechanisms.

It is also possible to derive statistics from such trace files, summarising the use of devices, and providing transaction latency histograms, for example. These can be used in sizing various required system resources and communications mechanisms.

6. JPEG ENCODING EXAMPLE

We applied this system-level design flow to a JPEG encoding example mapped onto a five-processor MPSoC system. Two of the processors acted as the source and sink for the JPEG examples, and thus served as the testbench for the system. Three processors were linked together in a dataflow style to form the core of the processing requirement, and the algorithm was divided into colour conversion, DCT and quantisation, and JPEG creation via Huffman encoding. Each processor had access to plentiful local and system level memory resources (these would be trimmed in a real system post-

analysis to the sizes required) and communicated with each other via direct HW FIFO queue implementations. Alternatively, experiments were run with shared memory implementations and a mixture of queues and shared memory. These experiments were run on a Pentium 4 based Linux workstation, running at 3.4 GHz with 1 GB memory.

Table 1: Fast vs. cycle accurate simulation for JPEG Encoding

Resolution	Sum of system cycles	Fast sim time (sec)	Sys. Cycle/ sec.	Cycle-accurate sim time (sec)	Sys. Cycle/ sec
32x32	636K	1.5	370K	22	29K
64x64	2.031 M	1.84	1.1 M	70.5	29K
128x128	21.452M	4.06	5.8 M	261	82K
256x256	85.522 M	9.55	9.0 M	1048	82K

Both fast functional simulation and cycle-accurate SystemC based simulation were used to validate the software and the system architecture. Table 1 illustrates, for a standard picture in 32x32, 64x64, 128x128 and 256x256 resolutions, the total number of simulated system cycles on the five processors, the elapsed time for the two simulations, and the corresponding system simulation rates. The simulated system used enormous HW queues as communications mechanisms for FIFO channels – 20K items deep. Significant processor stalling due to queues being full did not occur for the 32x32 and 64x64 resolutions. The table illustrates the difference between fast functional and cycle-accurate system simulations (the cycle-accurate simulation involved 75 system level devices – processors, local memories, system bus interfaces, routers, system memories, and hardware queue models). Further quick, hands-off optimisations were done on the configurable processors, using an automated tool but avoiding code restructuring, which reduced the number of cycles on the target processors by 18-26 %. Code restructuring that allowed vectorisation would give a further significant improvement in performance.

We also used statistical analysis of trace files to determine that a maximum queue depth for image data of 500 would allow the system to work without deadlock for all images, although further optimisation is possible.

7. CONCLUSION

The design of complex MPSoC systems poses many interesting and taxing challenges to system architects, SW and HW designers. This paper has outlined several of those challenges and developed a list of requirements for design technology that begins to offer such capabilities. As an example of what might be possible, an experimental MPSoC design tool was used to illustrate some of these technologies.

Many additional capabilities can be envisaged for such a design flow, including richer sets of abstract communications models, additional system level components, more automated tools for structural platform configuration, and automated mapping and analysis tools.

8. REFERENCES

[1] Ahmed Jerraya and Wayne Wolf (editors), *Multiprocessor Systems-on-Chip*, Elsevier Morgan Kaufmann, San Francisco, California, 2005.

[2] Chris Rowen and Steve Leibson. *Engineering the Complex SOC*. Prentice-Hall PTR, 2004.

[3] Steve Leibson and James Kim. "Configurable processors: a new era in chip design". *IEEE Computer*, July, 2005, pp. 51-59.

[4] Ruth Ivimey-Cook, "Legacy of the transputer", in B.M. Cook (editor), *Architectures, Languages and Techniques*, IOS Press, 1999.

[5] Richard Goering, "Multicore design strives for balance... but programming, debug tools complicate adoption", *Electronics Engineering Times*, March 27, 2006.

[6] Kang Su Gatlin, "Trials and tribulations of debugging concurrency", *ACM Queue*, Volume 2, Number 7, October 2004.

[7] Matthias Gries and Kurt Keutzer (editors). *Building ASIPs: The MESCAL Methodology*. Springer, 2005.

[8] Makiko Itoh, Shigeaki Higaki, Yoshinori Takeuchi, Akira Kitajima, Masaharu Imai, Jun Sato, and Akichika Shiomi, "PEAS-III: An ASIP Design Environment", *ICCD 2000*, pp. 430-436.

[9] Grant Martin, "ESL Requirements for Configurable Processor-based Embedded System Design", *IP-SoC 2005*, Grenoble, France, pp. 15-20.

[10] http://www.eclipse.org

[11] http://www.spiritconsortium.org

[12] Frank Ghenassia (editor), *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, Springer, 2005.

[13] P. van der Wolf, et. al., "Design and programming of embedded multiprocessors: an interface-centric approach", *CODES+ISSS 2004*, pp. 206-217.

[14] Pierre Paulin, et. al., "Application of a multi-processor SoC platform to high-speed packet forwarding", *DATE 2004*, Volume 3, pp. 58-63.