Model-based Programming Environment of Embedded Software for MPSoC

Soonhoi Ha

School of Computer Science and Engineering
Seoul National University
Seoul, 151-742, KOREA
Tel : 82-2-880-8382
Fax : 82-2-879-1532
e-mail : sha@snu.ac.kr

Abstract - A noble model-based programming environment of embedded software for MPSoC is proposed. By defining a common intermediate code (CIC), it separates modeling of the software and implementation optimized for target architecture. It also allows us to use diverse models for initial specification. Another feature is to provide multi-phase debugging capabilities: at the modeling stage, at the code generation stage, and at the simulation stage. Preliminary experiments with a Divx player confirm the feasibility and validity of the proposed technique.

I Introduction

Insatiable demand of system performance makes it inevitable to integrate more and more processing elements in a single chip, called MPSoC (Multi-Processor System on a Chip), to meet the performance requirement. While extensive research on the design methodology has been performed to aid the development of SoC, all but few, if any, focus on the design of hardware architecture. In reality, however, the bottleneck of MPSoC design will be not hardware design but software design, as a hardware platform is reused in platform based design. Thus this paper focuses on the software design methodology for MPSoC.

Compared with general purpose software, embedded software of MPSoC has the following characteristics.

(1) Concurrency: Embedded software for MPSoC is parallel programs on a multi-processor system that is typically a heterogeneous system including hardware components that run concurrently with software.

(2) Real-time constraints: An embedded system usually has real-time constraints. A typical solution to satisfy the real-time constraints is to use a priority-based real time scheduling of tasks. Since common scheduling algorithms such as RM (Rate-Monotonic) and EDF (Earliest-Deadline-First) algorithms assume a single processor system, they cannot be used directly for MPSoC. We need to develop a new way of checking the schedulability of tasks in an MPSoC.

(3) Resource constraints: Since the cost-performance ratio is a more important metric than performance alone, it is desirable to develop an optimized software that minimizes the resource requirement such as memory size.

(4) Verification: Since the fabrication cost of an MPSoC is huge and run-time debugging is not possible, it is critical to verify the software in a virtual prototyping environment before the chip is fabricated.

Recently, it becomes more popular to use a model driven architecture (MDA) for systematic design of software [1][2]. In an MDA, system behavior is described in a platform independent model (PIM). Based on the hardware platform specification, the PIM is translated to a platform specific model (PSM) from which the target software on each processor is generated. MDA methodology is expected to improve the design productivity of embedded software since it increases the reuse possibility of platform independent software modules: The same PIM can be reused for different target architectures. As the design cycle of MPSoC gets shorter and the time-to-market pressure continually increases, such methodology is also needed in the embedded software design of MPSoC. Thus we make the proposed technique also model-driven.

Unlike other model driven architectures, the unique feature of the proposed technique is to allow multiple PIMs in the programming environment. And we define a fixed form of PSM, called CIC (Common Intermediate Code), to which a PIM is translated after partitioning and mapping decision is made for a given target architecture. Since the CIC is independent of the communication architecture and OS of target system, we can explore some design space at the later stage of design. Another feature is to provide multi-phase debugging capabilities: at the modeling stage, at the code generation stage, and at the simulation stage.

We outline the proposed technique in the next section. Section 3 explains each design step with a real example, Divx player. The current status of implementation and some preliminary results will be discussed in section 4. We will draw conclusions in section 5.

II. Overview of the Proposed Methodology

Figure 1 shows the framework of the proposed embedded software development environment for MPSoC. In this section, 4 important design steps are overviewed.

A. Model-based programming

UML might be the most well-known model for embedded software design. UML-based software design tools, such as Telelogic TAU and IBM Rose-RT, have appeared on market. But the current UML-based tools do not satisfy the characteristics of embedded software, listed in the previous section: They do not generate parallel programs, nor check the schedulability of the generated software. And they do not consider the resource constraints at the stage of code generation nor provide verification capability of the generated software.

On the other hand many researchers advocate the use of actor-based models [3]. An actor is an active object that runs autonomously on the reception of input events or data samples. So it is recognized more suitable to express the concurrency of a system than a passive object in an object oriented model. In an object oriented model, an object is invoked by a method call from the outside. A well-known commercial tool, SIMULINK of The Mathworks inc. [4], is based on an actor model: And software generation from SIMULINK model is being adopted for software generation of automobile electronics [5][6].

While many models have been proposed for embedded software specification, no consensus is reached to any particular model that is good for all applications. Therefore we propose to allow diverse models in our programming environment instead of selecting one. In the current implementation, we use two models: UML 2.0 model and PeaCE model [7] as illustrated in **Fig. 1**. PeaCE model will be explained in the next section.

B. Common Intermediate Code (CIC)

Once an initial specification is made, the model is partitioned into multiple processors based on the architecture information. Since the optimal partitioning is beyond the scope of this paper, we assume that the partitioning results are given somehow. From the initial specification model and the partitioning results, we generate a common intermediate code (CIC) that is the key element in the proposed framework. Any model can be integrated to our framework if it can be translated into a CIC after partitioning. The CIC code is defined intuitively so that it can be designed manually.

The CIC contains the following information.
- partitioned tasks mapped to each processor and their codes
- communication and synchronization between tasks
- hardware information necessary for software generation such as address mapping of memory segments
- design constrains such as performance and power

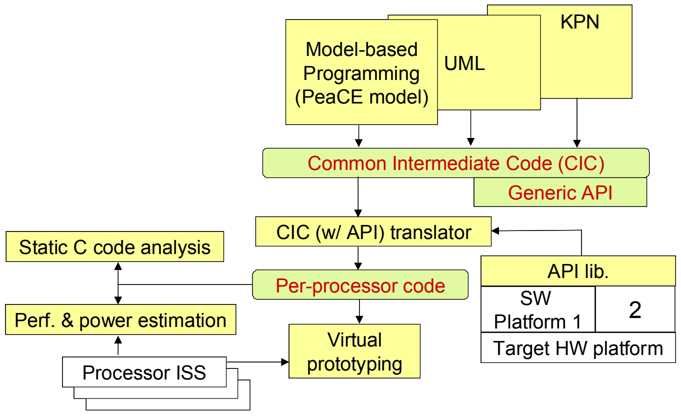


Fig. 1. The proposed embedded software development framework for MPSoC

The CIC uses generic API (Application Programming Interface) functions to express the I/O operation of tasks. Therefore the CIC is independent of the software platform and communication architecture. It means that the software platform in each processor can be determined independently.

Another feature of CIC is that it uses OpenMP specification [8] to express the data parallelism inside a task. We separate the specification of task parallelism and data parallelism. Task parallelism is accomplished by partitioning the specification model and by generating the partitioned tasks in each processor. Such separation is adequate especially for an MPSoC architecture that has a SIMD unit for data parallel executions.

C. Automatic Software Generation

The next step is to generate optimized software codes for each processor from the CIC. We first translate each task code into the final one by converting the generic APIs into OS APIs or communication APIs. If a task is run on a processor that has an OS, a generic API is translated to the corresponding OS API. Otherwise, it is translated to a communication API that is assumed to be optimized for the given target architecture. If the task has data parallelism with OpenMP pragmas, it is translated into a parallel program.

Another important issue is to schedule the mapped tasks in each processor based on the design constraints and optimization objectives. If a processor has no OS inside, a run-time system that schedules the tasks should be synthesized automatically.

D. Verification and Debugging

In the proposed framework, software is verified in three different phases. The first verification is performed in the modeling phase. The formality of each model enables us to detect the syntax error of the initial specification. The functionality of the program is verified by simulating the initial model if possible.

The second phase of verification is to analyze the generated code per processor in the source level. Static code analysis tools are integrated to the proposed framework to detect the possible error locations, particularly for memory access error such as buffer overrun, zero de-referencing, memory leak, etc.

To find the remaining errors and debug the program, we will provide the run-time debugging environment with a virtual prototyping system.

III. HOPES: The Proposed Framework

The proposed embedded software development framework, named HOPES, is under development. While it allows the use of any model for initial specification, the current implementation is being done with two models: PeaCE model and UML 2.0 model. In this section, we will explain the detailed procedure of the proposed technique, using the PeaCE model. PeaCE model is one that is used in PeaCE hardware-software codesign environment for multimedia embedded systems design [9].

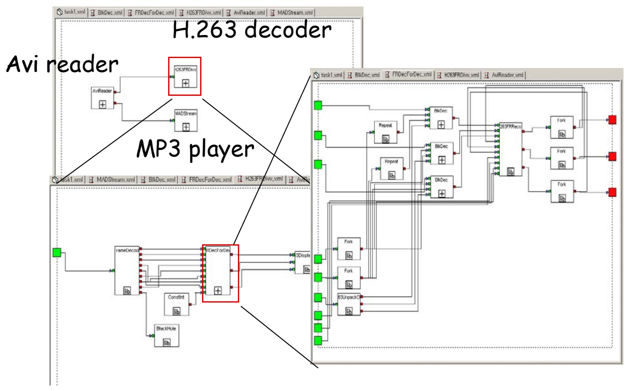


Fig. 2. Divx specification in PeaCE model

A. Programming in PeaCE Model

In PeaCE model, the system behavior is specified with a heterogeneous mixture of three different models of computation. At top level, the PeaCE model uses a task model that specifies the execution condition of each task and communication requirements between tasks. The internal definition of each task is specified with a formal model of computation. An extended SDF model, called SPDF model [10], is used to specify signal processing tasks or computation oriented tasks. The PeaCE model uses a hierarchical and concurrent FSM model, called fFSM [11], to specify the control tasks.

Fig. 2 displays an example of Divx specification in PeaCE model. At top model, three tasks are specified: Avi reader, MP3 player, and H.263 decoder. All three tasks are computation tasks of which the internal definition is specified in a hierarchical SPDF graph. The granularity of an atomic block is a reusable function such as IDCT and MC (Motion Compensation). In this example, no control task is specified.

Dataflow model is good for parallel programming model since it expresses only the true data dependency between function blocks so that parallelism inherent in the system behavior is explicitly shown. The SPDF model extends the SDF model in two ways: First, it can express dynamic control structure, such as if-then-else and for-loop, that is commonly used in most complex multimedia applications. Second, it allows shared data structure between function blocks, so the generated code can be made as memory efficient as manually optimized code.

In HOPES, the hardware architecture is separately specified in a block diagram. In an architecture diagram, the atomic block represents a processor or hardware component. Each block has some parameters that show the information necessary for software development. Using the architecture information, the initial specification of PeaCE model is partitioned into the processing components to generate the CIC. In this step, the number and the kinds of processing components and the approximate communication overhead are considered.

B. CIC Format

The CIC is an intermediate representation common to all initial specification models in the proposed environment.

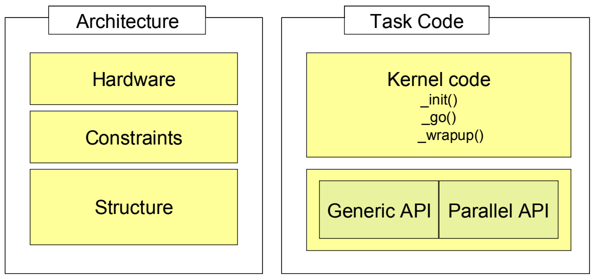


Fig. 3. CIC (Common Intermediate Code) Format

The CIC format consists of two parts, architecture parts and task code parts, as displayed in **Fig. 3**. The architecture part is further divided into three sections in an xml-style file. The “hardware” section contains the hardware architecture information that is necessary to generate the software. The “constraints” section specifies the real time constraints of tasks and the power consumption constraints if any. The “structure” section describes the communication and synchronization requirements between tasks. **Fig. 4** displays an example of the architecture part in the CIC that is automatically generated from the Divx specification in PeaCE model.

```
<?xml version="1.0" ?>
 <CIC_XML>
  <hardware>
   <processor name="arm926ej-s0">
     <index>0</index>
     <localMem name=“lmap0”>
        <addr>0x0</addr>
        <size>0x10000</size>  // 64KB
     </localMem>
     <sharedMem name=“shmap0”>
        <addr>0x10000</size>
        <size>0x40000</size>  // 256KB
        <sharedWith>1</sharedWith>
     </sharedMem>
     <OS>
        <support>TRUE</support>
     </OS>
   </processor>
    …
  </hardware>
```

(a) Hardware part

```
<constraints>
  <memory>16MB</memory>
  <power>50mWatt</power>
   <mode name="default">
   <task name="AviReaderI0">
     <period>120000</period>
     <deadline>120000</deadline>
     <priority>0</priority>
       <subtask name="arm926ej-s0">
       <execTime>186</execTime>
       </subtask>
     </task>
     <task name="H263FRDivxI3">
       <period>120000</period>
       <deadline>120000</deadline>
        …
      </task>
   </mode>
 </constraints>
```

(b) Constraints part

```
<structure>
  <mode name="default">
    <task name="AviReaderI0">
      <subtask name="arm926ej-s0">
      <procMap>0</procMap>
      <fileName>AviReaderI0_arm926ej_s0.cic
                         </fileName>
      </subtask>
    </task>
    <task name="H263FRDivxI3">
      <subtask name="arm926ej-s0">
      <procMap>0</procMap>
      <fileName>H263FRDivxI3_arm926ej_s0.cic
                         </fileName>
      </subtask>
    </task>
     </subtask>
   </task>
  </mode>
```

```
  <queue>
    <name>mq0</name>
    <src>AviReaderI0</src>
    <dst>H263FRDivxI3</dst>
    <size>30000</size>
  </queue>
  <queue>
    <name>mq1</name>
    <src>AviReaderI0</src>
    <dst>MADStreamI5</dst>
    <size>30000</size>
  </queue>
</structure>
</CIC_XML>
```

(c) Structure part

Fig. 4. An example of the architecture part of an CIC

As shown in **Fig. 4**, the hardware section defines the address range and the size of each memory segment. The processor indices for the shared memory segments indicate which processors share the segments. The constraints section shows the global constraints such as power consumption and memory requirement and the per-task constraints such as period, deadline, and priority. And it also includes the estimated execution time of the tasks. Using these information, we will determine the scheduling policies of the target OS or synthesize the run-time system for the processor without OS.

In the structure section, two methods of task communication are supported: message queue and shared memory. In this example, only message queue is used for inter-task communication. It also indicates the file name (with ".cic" suffix) where the generated code of the task is contained.

As depicted in **Fig. 3**, each task code consists of three functions: {task name}_init(), {task name}_go(), and {task name}_wrapup(). The {task name}_init() function is called once when the task is invoked to initialize the task. The {task name}_go() function defines the main body of the task and is executed repeatedly in the main scheduling loop. The {task_name}_wrapup() method is called before stopping the task to reclaim the used resources.

In the proposed framework, these three functions should be automatically generated from the initial model and partitioning result. And estimated execution time of each partitioned task and inter-task communication requirements should be also written into the CIC.

Fig. 5 shows a code segment of "h263decoder_go()" function as an example CIC task code. Note that it uses a generic API, MQ_RECEIVE, for inter-task communication and an OpenMP pragma to express data parallelism inside the task. The OpenMP translator will translate the code to invoke the same number of threads as the number of processors to run them concurrently.

C. Automatic Software Generation: CIC Translator

The CIC does not take into account of the architecture details. The architecture details are considered when the CIC is translated into the final codes that are optimized for the target architecture. The CIC translator consists of three parts. First each task code in a CIC is translated into a parallel program via OpenMP translator if data parallelism exists in the task code. The number of available processors for data parallelism is given from the architecture specification.

```
void h263decoder_go (void) {
            ...
      l = MQ_RECEIVE("mq0", (char *)(ld_106->rdbfr), 2048);
      ...
      # pragma omp parallel for
      for(i=0; i<99; i++) {
          //thread_main()
              ....
      }
       // display the decode frame
         dither(frame);
}
```

Fig. 5. An example of CIC task code

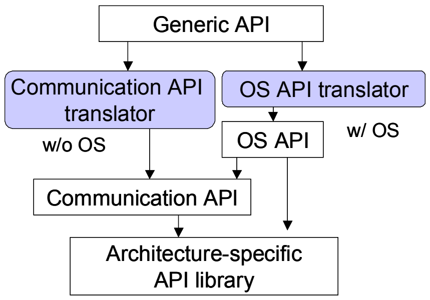


Fig. 6. Generic API translation

Depending on the target architecture, a different style of parallel program is generated. For a shared memory architecture, the OpenMP translator generates a multi-threaded code while it generates an MPI program for a distributed memory architecture.

The second part of the CIC translator is to translate the generic APIs into target-specific APIs and the last part is to synthesize the run-time system or to determine the scheduling policy in each processor depending on the existence of OS.

Determining what kind of OS and which OS to use is another important design decision in an MPSoC. One choice is to use an OS of master-slave structure. The master OS sits on a master processor and controls the slave OSes in other processors. The slave OS relies on the master OS for heavy OS-specific function except for scheduling the tasks mapped to the processor. Or we may want to use a single OS on a master processor and run the other processors without OS installed. Then the other processors play the role of co-processors of the master processor to accelerate some time-critical computation. Another option is to install a separate OS in each processor to make it a distributed system as a whole. Or we may want to use a multiprocessor OS.

Since an MPSoC is typically a heterogeneous system, the master-slave structure or co-processor style of operating system is likely to be used for simple implementation. Considering these possibilities, we define the CIC to be independent of the OS structure by using the generic APIs. We define about 70 API functions that are abstracted from IEEE POSIX 1003.1-2004 standard APIs and standard C library functions, carefully selecting the most heavily used. The programmer should use generic APIs for file access, I/O, inter-task communication, and synchronization. Some APIs may be used only in the processor with OS installed. This constraint should be considered when the partitioning decision is made.

Generic APIs are translated into target specific APIs via the procedure as shown in **Fig. 6**. If the target processor has an OS installed, generic APIs are translated into OS APIs. Otherwise they are translated into communication APIs. Communication APIs are defined directly accessing the hardware devices. We implement the OS API library and communication API library optimized for each target architecture.

Fig. 7 shows the internal structure of the "OS API translator" by which a generic API is translated into an OS API [12]. The inputs to the translator are a CIC code, pattern information and parameters for each generic API, and the file that describes the translation rule. The pattern of an API depicts the typical usage of the API in the code.

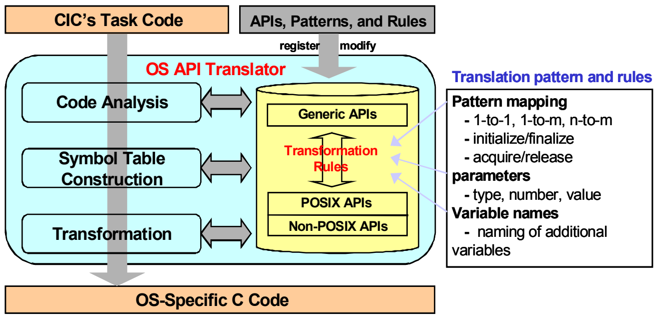


Fig. 7. Internal structure of OS API translator

In some cases, the pattern is defined for a pair of APIs: for example synchronization APIs using a semaphore. If a new API is defined, the pattern information, parameters, and the translation rule need to be added to the inputs. The OS API translator analyzes the CIC code, detects the generic APIs, constructs the symbol table, and performs translation.

After generic APIs are translated, the optimized code is generated for each task. The remaining work is to create a run-time system that schedules the tasks assigned to each processor considering the real-time constraints. If there is an OS, priority-based scheduling can be used. Otherwise, a customized run-time system is automatically synthesized.

D. 3-Phase Verification

Software verification is critical to improve the design productivity of MPSoC because it is not possible to debug and correct the software after the chip is fabricated. Therefore it is an important goal to detect as many errors as possible through 3 phase verification in the proposed environment.

The first phase is the modeling phase. We expect that the modeling front end provides the capability to detect the modeling error. For example, the PeaCE modeling front end detects some semantic errors in FSM and dataflow specification as well as syntax errors: Deadlock or buffer overflow errors are detected in a dataflow program and determinacy and reachability are tested in an FSM program. In this step, we assume that function blocks have no errors inside. Then, the CIC code is also assumed correct since the CIC is automatically generated from the model,

The second phase is to statically analyze each task code that is translated from the CIC on each processor. It is not easy to debug an embedded software with a run-time debugger if errors exist in the interface modules with the outside. Therefore we provide a static analysis tool that can detect buffer overrun error, memory leak, zero de-referencing, and stack overflow error [13][14]. The static analyzer reports all possible error locations so that the programmer examines the locations to manually detect and correct the errors. Performance of the static analyzer is measured by the speed and the false alarm ratio. In this phase, we detect the memory-related errors that exist inside function modules.

The final phase is to detect the run-time error by running the program on a virtual prototyping system. A virtual prototyping system is a simulation environment that mimics the real behavior of hardware components with simulation models. Commercial tools, such as ConvergenSC, MaxSim, and Seamless CVE, have been successfully used and proved its usefulness to develop software without building a real hardware prototype. The main target of those tools is an SoC simulation with a single processor core. While they can be used for multi-core systems, they have limitation on simulation speed and extensibility. Moreover, they do not provide the debugging capability of embedded software. In our research, we are developing a distributed simulation environment with parallel debugging capability. We aim to increase the simulation speed by parallel co-simulation.

In addition, we are developing a tool to analyze the performance and power of the system. The tool gives a detailed report of estimated performance and power for each function in the generated C code. The programmer may find out the candidate functions that need further optimization from the report.

IV. Preliminary Experiments

A. Implementation

The proposed programming environment is being developed on a Linux platform while the graphical user interface is developed on a window platform. The GUI and the engine are connected by TCP/IP socket. The environment consists of a few tens of software modules that are being developed separately. The interface between software modules is defined by files.

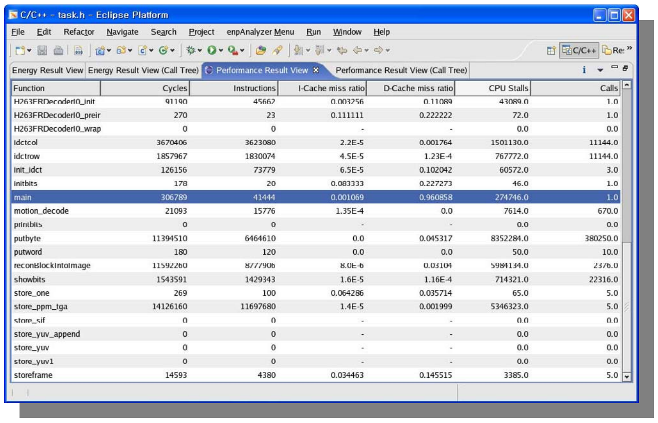
The current status of implementation is as follows: the design flow is established and software modules have been developed for a single processor target. The design flow is demonstrated with a Divx player example on a single ARM processor, starting with a PeaCE model representation. On the other hand, we have defined a new UML 2.0 subset, called ESUML (Embedded Systems UML), that is a light-weight UML that consists of only 5 diagrams for embedded software specification.

Software modules are now being extended for multi-processor targets. Partitioning algorithms are being developed for PeaCE model and ESUML model separately. OpenMP translator and virtual prototyping system with parallel debugging capability are being developed.

B. Design Flow Demonstration with a Divx Player Example

A Divx player application that consists of three dataflow tasks is specified in a PeaCE model as already shown in **Fig. 2**. To confirm the 3-phase verification capability, we inserted three different types of errors. First we modified the ratio of decoding rate between Y, U, and V frame in H.263 decoder task to 3:1:1 from 4:1:1. This modification generates a buffer overflow error on the input arc of Y frame decoding module. This error was successfully detected at the modeling stage since the PeaCE modeling module analyzes the SPDF specification to detect any sampling inconsistency error between function modules.

Second, we inserted an error inside the "Avi Reader" task: We decrease the array size to generate a buffer overrun error. Our static analyzer, called Airac 5[14], detected the buffer overrun error successfully. Lastly, we inserted a logical error in the H.263 decoding task. The logical error could be detected with the ARM simulator by manual debugging.


C/C++ - task.h - Eclipse Platform
File Edit Refactor Navigate Search Project enpAnalyzer Menu Run Window Help
C/C++
Energy Result View
Energy Result View (Call Tree)
Performance Result View
Performance Result View (Call Tree)

Function	Cycles	Instructions	I-Cache miss ratio	D-Cache miss ratio	CPU Stalls	Calls
H263FRDecoderI0_init	91190	45662	0.003256	0.11089	43089.0	1.0
H263FRDecoderI0_preir	270	23	0.111111	0.222222	72.0	1.0
H263FRDecoderI0_wrap	0	0	-	-	0.0	0.0
idctcol	3670406	3623080	2.2E-5	0.001764	1501130.0	11144.0
idctrow	1857967	1830074	4.5E-5	1.23E-4	767772.0	11144.0
init_idct	126156	73779	6.5E-5	0.102042	60572.0	3.0
initbits	178	20	0.083333	0.227273	46.0	1.0
main	306789	41444	0.001069	0.960858	274746.0	1.0
motion_decode	21093	15776	1.35E-4	0.0	7614.0	670.0
printbits	0	0	-	-	0.0	0.0
putbyte	11394510	6464610	0.0	0.045317	8352284.0	380250.0
putword	180	120	0.0	0.0	50.0	10.0
reconBlockIntoImage	11592260	8777906	8.0E-6	0.03104	5984134.0	2376.0
showbits	1543591	1429343	1.6E-5	1.16E-4	714321.0	22316.0
store_one	269	100	0.064286	0.035714	65.0	5.0
store_ppm_tga	14126160	11697680	1.4E-5	0.001999	5346323.0	5.0
store_sif	0	0	-	-	0.0	0.0
store_yuv_append	0	0	-	-	0.0	0.0
store_yuv	0	0	-	-	0.0	0.0
store_yuv1	0	0	-	-	0.0	0.0
storeframe	14593	4380	0.034463	0.145515	3385.0	5.0

Fig. 8. Performance analysis result: H.263 decoder

We analyzed the performance of the system as illustrated in **Fig. 8** where per-function information is reported on the CPU cycle, number of instructions, cache misses, and so on. The same information can be obtained in the form of call tree.

V. Conclusions

Embedded software development is very challenging task in an MPSoC design particularly because it is parallel programming in nature and its verification should be done before building a chip. Software design environment for general purpose software can not be used since they usually do not consider the design constraints that embedded software should satisfy: real-time constraints, resource constraints, and concurrency. Moreover making a parallel program itself is not an easy task.

In this paper, we proposed a new design environment, called HOPES, that aids to develop embedded software for MPSoC. It starts with a model-based specification that is independent of the target architecture. By defining a common intermediate code (CIC), it accommodates diverse models at the front end. We explained how the CIC code is finally translated into the software core on each processor. The proposed environment also provides 3-phase verification to detect as many design errors as possible. It was shown through preliminary experiments that the proposed 3-phase verification technique could detect various kinds of errors.

While the proposed technique is mainly targeted for MPSoC, it can be used for software development of any embedded system. Currently, only the feasibility of the design flow was proved with a Divx example on a single processor core. We expect that successful implementation of the proposed environment will improve the design productivity of MPSoC significantly.

Acknowledgements

This work was supported by IT leading R&D Support Project funded by Korean MIC and BK21 project. The ICT and ISRC at Seoul National University provided research facilities for this study. I would like to express special thanks to all team members who are involved in the HOPES projects.

References

[1] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003.

[2] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing applications using model-driven design environments," *IEEE Computer*, Vol. 39(2), pp. 33-40, 2006.

[3] Edward A. Lee and Stephen Neuendorffer, "Concurrent models of computation for embedded software," IEE Proceedings, Computers and Digital Techniques, Vol. 152, Issue 2, pp. 239-250, 2005.

[4] SIMULINK, The MathWorks Inc. (http://www.mathworks.com)

[5] dSPACE, dSPACE inc. (http://www.dspaceinc.com)

[6] Real-Time Workshop 6.5, The MathWorks Inc.

[7] Kiseun Kwon, Youngmin Yi, Dohyung Kim, Soonhoi Ha, "Embedded software generation from system level specification for multi-tasking embedded systems", *ASP-DAC`05,* Vol. 1 pp 145-150, 2005

[8] OpenMP C and C++ API, version 1.0, http://www.openmp.org, 1998.

[9] Soonhoi Ha, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo, "Hardware-software codesign of multimedia embedded systems: the PeaCE approach", The *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Vol. 1 pp 207-214, 2006.

[10] Chanik Park, Jaewoong Chung and Soonhoi Ha, "Extended synchronous dataflow for efficient DSP system prototyping", *Design Automation for Embedded Systems*, Kluwer Academic Publishers Vol. 3 pp 295-322, 2002.

[11] Dohyung Kim, Soonhoi Ha, "Static analysis and automatic code synthesis of flexible FSM model", *ASP-DAC*, pp. 161-165, 2005.

[12] J. Maeng, J.H. Kim, and M. Ryu, "An RTOS API translator for model-driven embedded software development," *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)* pp. 363-367, 2006.

[13] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi, "Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis," *Lecture Notes in Computer Science*, Vol.3672, pp.203-217, 2005

[14] Airac5, (http://ropas.snu.ac.kr/airac5)