Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software

Lovic Gauthier Sungjoo Yoo Ahmed A. Jerraya

SLS Group, TIMA Laboratory
46 Avenue Felix Viallet, 38031 Grenoble, France
{Lovic.Gauthier,Sungjoo.Yoo,Ahmed.Jerraya}@imag.fr

Abstract

We propose a method of automatic generation of application specific operating systems (OS's) and automatic targeting of application software. OS generation starts from a very small but yet flexible OS kernel. OS services, which are specific to the application and deduced from dependencies between services, are added to the kernel to construct the whole OS. Communication and synchronization functions in the application code are adapted to the generated OS. As a preliminary experiment, we applied the proposed method to a system example called token ring system.

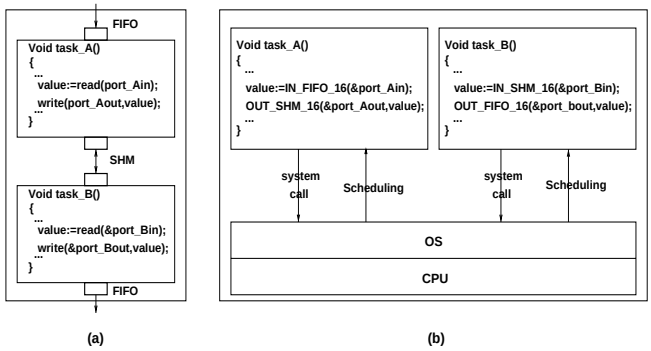

Figure 1. An example of OS-based SW implementation of multiple tasks.

1. Introduction

SW parts of embedded systems are taking more and more system resources in terms of numbers and sizes of processors, memory usage [9] or power consumption [4]. To implement complex SW on the target processors, operating systems (OS's) are usually adopted to serialize SW execution and to interface SW application to the target architecture[1].

Figure 1 exemplifies OS-based SW implementation of concurrent multiple tasks on a processor. Figure 1 (a) shows two concurrent tasks that communicate with other tasks via high abstraction level channels like FIFO and shared memory. In the OS-based SW implementation, the OS schedules the execution of tasks and executes communication between tasks via system calls. Figure 1 (b) shows a case of OS-based SW implementation of the tasks. In the figure, two tasks A and B communicates with other tasks via system calls (in this case, IN_FIFO_16, IN_SHM_16, OUT_FIFO_16, and OUT_SHM_16).

[1]In this paper, the target architecture represents processor(s), memory modules, devices, etc.

One of crucial problems in such OS-based SW implementation is that porting or configuring the OS on the target architecture (the target processor and memory architecture) and targeting SW application code on the OS are mostly done by manual work, i.e. the designer ports the OS (sets OS configurations) on his/her specific target architecture and modifies the application SW code to meet the ported/configured OS. Such a design practice is time consuming and error prone. Moreover, change of target architecture can require significant re-porting/re-configuration of OS and, possibly, re-targeting of SW application code. Thus, in such a design practice, finding the optimal target architecture and OS configuration, i.e. **design space exploration (DSE)** of OS and OS-related target architecture, seems to be hard within the even tighter time-to-market.

To enable DSE of OS implementation, methods for automatic generation of application specific OS's are required. In this paper, we present a method that gives automatic generation of application specific OS's and automatic targeting of the application code to the generated OS.

This paper is organized as follows. We give a review of

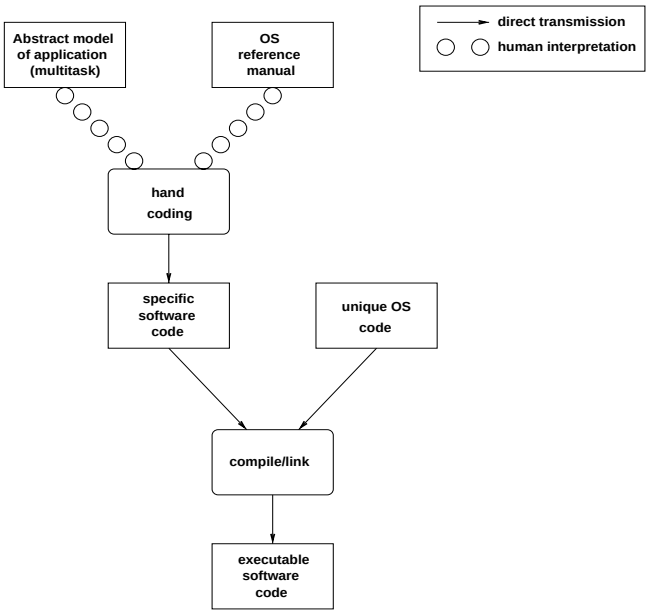

Figure 2. SW implementation with the ported OS.

related work in Section 2. We propose a method of automatic generation of application specific OS and automatic targeting of application SW code in Section 3. In Section 4, we present a case study of applying the proposed method. In Section 5, we give conclusion.

2. Related Work

In this Section, we review previous studies on (1) implementing concurrent multiple tasks on a single processor and (2) application specific OS generation.

There are three approaches in SW implementation from multi-task descriptions. The first two approaches use OS as a scheduler and an interface of multiple tasks to the target architecture.

1. The designer ports existing OS's on the target architecture, targets multiple tasks to the OS's, and runs them on the OS's. Figure 2 shows the OS porting and SW targeting flow in this approach.

2. To improve the performance of OS (e.g. system call execution times) and/or to reduce the overhead of OS (e.g. OS code size), the designer can configure existing OS's [12][6]. For instance, the designer can determine the usage of message queue in the OS depending on whether the application SW uses it or not. The granularity of such a configuration depends on OS vendors.

3. To prevent the usage of OS, a sequential code can be generated from the concurrent multi-task representation [5]. In this approach, there is a trade off between the reduction of OS overhead and the increase of sequential code overhead in terms of code size, system runtime, etc.

Table 1. Comparisons of different approaches of multi-task SW implementation.

Approaches	speed	size	flexibility	efforts
OS Porting	$***$	$*$	$*$	$*$
OS Configuration	$***$	$**$	$**$	$**$
Sequential code gen.	$*$	$*$	$**(*)$	$***$

Recently, there are a few approaches that mix sequential code generation (i.e. maximizing sequential code parts in the application) and (then) OS usage [2][10]. To enable OS-based system design at a higher abstraction level, a high level model of OS called SoCOS is presented in [3]. In [8], an analysis is given on context switching overhead of three context switching methods (pico-kernel, code merging and table-based sequencing) in non-preemptive task scheduling.

Table 1 compares the above three approaches in terms of OS execution time (speed), memory requirement (size), scalability and portability (flexibility), and designer's efforts (efforts). In the table, three (two and one) stars represent an excellent (average and poor) note. In the case of OS configuration, we put two stars in size, flexibility and efforts since there are several configurable OS's, but their quality (degree of configuration) differs from one to another. In the case of sequential code generation, the reasons of poor note in speed are (1) synchronization can be done only by polling[2] and (2) a lot of conditional jumps in the sequential code do not fit well with the pipelines and caches of the state-of-the-art processors. Note that in the above-mentioned approaches of OS-based SW implementation, targeting the application SW on the ported/configured OS still requires a lot of hand coding.

In our method proposed in this paper, the OS is generated, from a very small and flexible OS kernel, including only the application specific functions. Compared with the OS configuration approach, the proposed method can give more efficient adaptation of OS to the application SW since determining application specific functions is done automatically and we start OS generation with a very small and flexible OS kernel.

In terms of adapting the OS to the specific application, the proposed method is related with (1) the work in [11][10] and (2) a method called **OS specialization** [1]. In [11], the authors present a strategy of automatic targeting of communication in OS's (based on a parameterized communication

[2]In physical implementation, interrupt can be used to propagate the synchronization event. However, detection of such an event is done by polling the event.

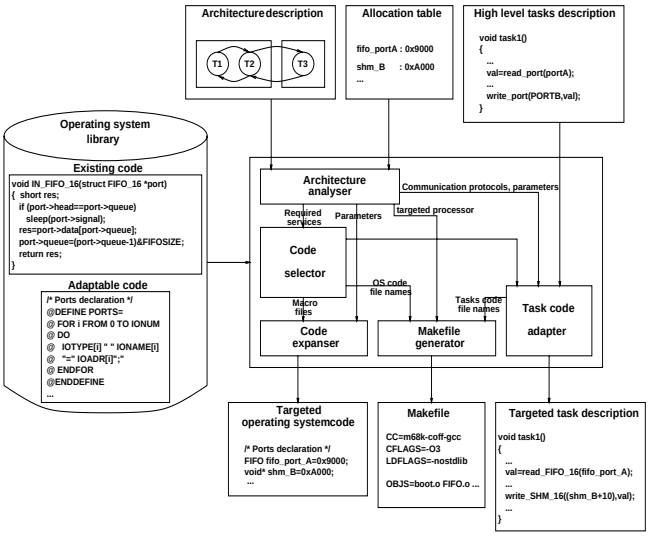

Figure 3. A flow of automatic generation of application specific OS and automatic SW targeting.

library and a simple scalable target architecture) with already targeted OS's. In [1], an automatic synthesis method of task scheduler is presented. Compared with the work in [11][10], the proposed method automates both (1) the generation of the whole OS (i.e. scheduler and inter-task communication implementation) specific to the application and target architecture (in a systematic construction of OS with application-specific and derived OS services) and (2) the targeting of high-level inter-task communication of the application to the generated OS. The difference between the proposed method and the OS specialization method is that we focus on generating the OS with the minimum and sufficient services required in the application while the OS specialization methods optimize OS services themselves exploiting quasi-static behavior of OS in the specific application.

Our method is also very similar to the one proposed in [7]. The main difference is the application domain: while their method mainly focus on dataflow application, our ambition is to focus on more heterogenous application domain. Due to such a difference the input model of the application changes and the OS library has to be more complex (see Section 3.3 and Section 3.4).

3. Automatic Synthesis and Targeting of Application Specific OS's and Application SW Codes

3.1. Design Flow for SW Implementation

Figure 3 shows our design flow of automatic OS generation and SW targeting. The input to the OS generation

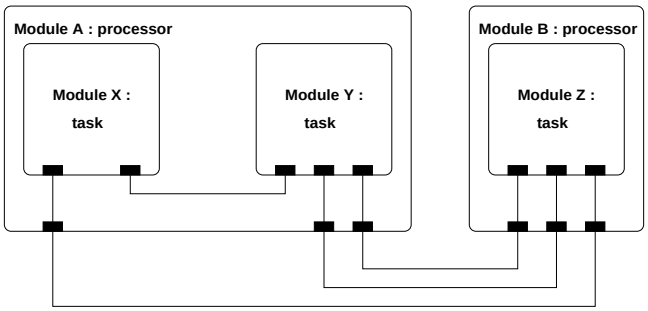

Figure 4. Hierarchical network of modules.

and SW targeting flow is a system description consisting of hierarchically structural representation of communication between modules, module behavior, and memory allocation information. As shown in the figure, Architecture Analyzer takes the structural information and the memory allocation information. Code Selector receives a list of services specific to the application from Architecture analyzer and finds the full list of (original and deduced) services, Code Expander generates the source code of OS. Task Code Adapter performs SW targeting and Makefile Generator gives makefiles. Thus, the outputs of the design flow are the source code of generated OS, targeted application code, and a makefile for each processor. To obtain the binary code to be downloaded onto the target processor memory, the designer runs compilation of both generated OS and targeted application using the generated makefile.

3.2. Application domain

We focus our targeting tool on heterogenous embedded applications (e.g. cellular phones, car controllers, etc.). These applications require various communication protocols, and may have very different time constraints (even within the same processor).

3.3. System Description Input

As the system description input, the flow takes (1) a structural representation of communication in a hierarchical network of modules, (2) behavior code of tasks and (3) a memory allocation table.

Figure 4 shows an example of hierarchical network of modules. In the structural representation, **modules** are connected via **communication channels**. In the hierarchical representation, each module can be a leaf module or a module that has a network of modules inside of it. We call a leaf module a **task**. A module consists of **behavior** and **port(s)** i.e., in the representation, behavior and communication are separated. The behavior part uses the ports (via calling port functions) to communicate with other modules. In

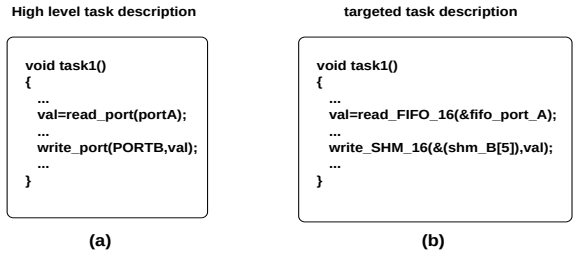

Figure 5. An example of task behavior.

the viewpoint of behavior, ports give high-level communication functions encapsulating communication details (i.e. communication protocols). Figure 5 (a) shows an example of communication via ports in a task. In the figure, the behavioral part just calls a high-level communication function (i.e. a port function), **read_port(portA)** for communication via port **portA**.

SW targeting should refine calls to port functions. Figure 5 (b) exemplifies a case of SW targeting of the task description in Figure 5 (a). In the figure, port function read_port(portA) is replaced by a system call, read_FIFO_16(fifo_port_A) of the generated OS.

In the structural description, **the information of behavior mapping** on the multi-processor target architecture is also given Figure 4 that shows each module (in this case, task) is mapped to processor A or B. The OS is generated on a processor basis. For OS scheduling, each task has a **task priority**. Mapping information and task priority are represented with attributes of the module.

In the memory allocation table, the information of memory allocation for inter-processor communication (e.g. memory allocation for fifo or shared memory for inter-processor communication) is given.

3.4. Operating System Library

The OS library provides (1) a very small and flexible OS kernel and (2) OS services.

3.4.1 OS Kernel

The main functionality of OS kernel is scheduling multiple tasks. There are several preemptive schedulers available in the OS library such as round-robin scheduler, priority-based scheduler, etc. In the case of round-robin scheduler, time-slicing (i.e. assigning different CPU load to tasks) is supported. To make the OS kernel very small and flexible, (1) the task scheduler can be selected from the requirement of the application code and (2) a minimal amount (less than 10% of kernel code size) of processor specific assembly code is used (for context switching and interrupt service routines).

Table 2. Basic functions of OS kernel.

Function	Behavior	Code type
ContextSwitch	Context switching	C + assembly
Sleep	Task sleeping	C
Wake-up	Waking up the task	C
Schedulers	Task scheduling	C

Table 2 shows the basic functions of OS kernel. Function **ContextSwitch** performs context switching between the currently running task and the next task to be executed. Since context switching operation differs from processor to processor, the function consists of two kinds of code: C code and assembly code. The C code part is called by other schedulers in C code (e.g. by a priority based scheduler or a round-robin scheduler). The assembly code part performs processor specific context switching operation. Function **Sleep** and **Wake-up** are used for preemptive task scheduling. For task scheduling, schedulers required by the application SW are selected from the OS library (as scheduler services).

3.4.2 OS Services

The OS library provides services specific to embedded systems: communication services (e.g. fifo communication), i/o services (e.g. PCI bus drivers), memory services (e.g. cache or virtual memory usage), etc.

Each service may be provided by one or more OS element described into the library. OS elements represent some part of the OS. They provide some services, and may require some other services. They may have several implementations compatible with different architectures.

Implementations of the OS elements contain two types of code: re-usable (or existing) code and expandable code. As an example of existing code, a fifo code can exist in the OS library in the form of C language. As an example of expandable code, OS kernel functions can exist in the OS library in the form of macro code. In Figure 3, examples of existing and expandable codes are shown in the OS library. In Section 3.5.3, we explain the code expansion in detail.

3.5. OS Code Generation

3.5.1 Architecture Analyzer

Architecture Analyzer finds the following information from the system description input.

- Application specific services and their detailed parameters
- Module specific parameters

Application specific OS services are found from the attributes of modules, channels and ports in the system description input. For instance, if a channel has an attribute for fifo implementation, fifo service is selected to be included into the OS to be generated. The detailed parameters of required services are also found from the allocation table. For instance, the address range of fifo communication and the interrupt priority of interrupt-driven port can be found from the allocation table. The information of required services is sent to Code Selector.

Module specific parameters (e.g. task priority, CPU load, the type of mapped processor, etc) are also found from module attributes. The type of processor is sent to Makefile Generator to choose the right compiler and to Code Expander to target the OS code to the processor.

3.5.2 Code Selector

Code Selector takes as input a list of required services from Architecture Analyzer. It looks up the OS library to check service dependencies and finds all the **elements** that have dependency relation with the required services and that are compliant with the target architecture. An element is compliant with an architecture if one of its implementations is compliant with the architecture, and if all the services it requires can be provided by a compliant element. Since an element may require some services, the above algorithm repeated recursively. For instance, a required service, fifo communication should need an interrupt handling service. In this case, an element providing the interrupt handling service should be also chosen to be included to the OS to be generated. Sometimes, several elements compliant with the architecture may provide the same required service. In such a case, it is up to the user to choose the good one.

After the element selection is done, Code Selector sends the list of the code file names to Makefile Generator, and the macro file names to Code Expander.

3.5.3 Code Expander

Code Expander takes as input a list of macro code from Code Selector and parameters (processor and allocation informations) from Architecture Analyzer. It generates the final OS code by expanding the macro codes of elements to source codes (in C or assembly).

Figure 6 shows an example of code expansion. In Figure 6 (a), a macro code Section is shown for two OS kernel functions: a context switch function (in the figure, **ContextSwitch()**) and a round-robin scheduler function (**Circle()**). First, Code Expander determines the necessity of services based on the information of requested services. For instance, in Figure 6 (a), depending on the number of priority values (in the figure, **Pr_max**) and the number of

```
@DEFINE schedule=IF ((Pr_max>0)||(G_size>1)) DO"
@void ContextSwitch()
@{
@      int oldtid=curtid;
@      if (switching)        {
@            "getactivetask"
@            "taskswap"
@      }
@}
@"        ENDIF
@ENDDEFINE
 @schedule@

@DEFINE round_robbin=IF (G_size>1) DO "
@void Circle()
@{
@      arciclepos=circlepos;
@      circlepos=circletabl[circlepos];
@}
@"        ELSE "" ENDIF
@ENDDEFINE
 @round_robbin@
```

(a) Macro code example

```
void ContextSwitch()
{
      int oldtid=curtid;
      if (switching)          {
            curtid=circlepos;
            taskswap_68k(tasks[oldtid].cxt,
                        tasks[curtid].cxt);
      }
}

void Circle()
{
      arciclepos=circlepos;
      circlepos=circletabl[circlepos];
}
```

(b) Expanded code example

Figure 6. An example of macro code expansion.

tasks (**G_size**) that have the same priority value, the context switch function (**ContextSwitch()**) or the round-robin scheduler function (**Circle()**) can be selected or not. In the example, if there is only one priority value (i.e. **Pr_max**=1) and **G_size**=1, then there is no need of context switching. Thus, in the case, the context switching code (**ContextSwitch()**) is not selected. If there are more than one tasks that have the same priority value, i.e. **G_size**>1, then the context switching code (**ContextSwitch()**) as well as the round-robin scheduler code (**Circle()**) are selected to be expanded. Figure 6 (b) shows an example of expanded code in C for this case. Note that, in this case, another scheduler can be selected to schedule tasks that have different priority values.

Figure 6 shows an example of expanding a macro, "`taskswap`" (in Figure 6(a)) to a processor specific code, "`taskswap_68k`" (in Figure 6(b)). Processor-specific code expansion is limited to functions such as context switching, synchronization primitives (e.g. semaphore functions), and interrupt service routines.

3.6. Targeting of Application SW Code

To target the application SW code to the generated OS, Task Code Adapter replaces function calls of communication and synchronization in the original application SW by OS service calls (i.e. system calls). For instance, function call of communication called **read_port(portA)** is replaced by a OS service function call
read_FIFO_16(&fifo_port_A). Note that original function arguments are also replaced by arguments specific to the OS service function. In the example, the argument of original

function **portA** is replaced by an argument of fifo service function **&fifo_port_A** that is a pointer to the fifo address in this case.

3.7. Makefile Generator

Makefile Generator takes as input (1) processor type information from Architecture Analyzer, (2) a list of source codes of OS (in C and assembly) from elements of Code Selector and (3) a list of the application SW codes. It determines the right compiler and linker and generates a makefile (for each processor) that includes the two code lists of OS and application SW.

3.8. Application to Existing OS's

The existing OS's can be integrated into the proposed flow of automatic generation (to be specific, in this case, automatic configuration) of application specific OS. To explain the integration, we assume that the existing OS supports OS configuration by `#ifdef` statements (i.e. configuration by defining required macros) without modifying the OS source code since most commercial OS's allow such a configuration. The integration can be done as follows.

1. Information of available OS services (e.g. service functions, macros to be defined for services, etc.) and dependency relations between services of the existing OS are taken into the OS library.

2. To the OS generation flow in Figure 3, the designer gives the same system description input as explained in Section 3.3.

3. Architecture Analyzer performs the same operation (extracting required information such as services, target processor information, etc) as described in Section 3.5.1.

4. Code Selector finds all the required (derived) services from the OS library as explained in Section 3.5.2. Then, it selects, from the OS library, macro definitions corresponding to the required services instead of selecting existing/adaptable files as explained in Section 3.5.2. Note that in the case of automatic configuration of existing OS, Code Expander does not generate the OS source code since the existing OS source code is not modified.

5. Task Code Adaptor performs the same operation (as explained in Section 3.6) for the automatically configured OS.

6. Makefile Generator outputs a makefile with the selected macro definitions received from Code Selector.

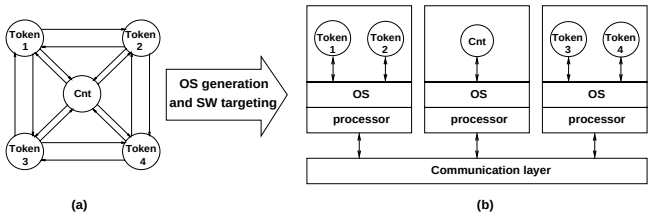

Figure 7. An implementation of token ring system on a multi-processor target architecture.

Note that compared with the original flow of automatic OS generation proposed in this paper, in the case of integrating the existing OS into the flow, there is no change in automatic execution of service extraction (by Architecture Analyzer), makefile generation, and adaptation of application code to the automatically configured OS. In terms of code quality (size, execution time, etc) of automatically configured OS, it depends on the granularity of OS services in the existing OS. Thus, if the existing OS supports as fine granularity in OS services as our OS kernel and services, the quality of automatically configured OS can be comparable to that of automatically generated OS.

4. Experiment

4.1. System Example and Target Architecture

We applied the proposed method to a system example called token ring system (1,245 lines in SystemC). It consists of four tasks (called Token) that exchange tokens with each other and one counter task (called Cnt) that counts the number of tokens exchanged. Figure 7 (a) shows the interconnection of tasks (in this example, a task corresponds to a module in the figure) in the example. As shown in the figure, four Token tasks make a bidirectional ring connection with each other. The counter task is connected to all Token tasks. Note that the structural description in Figure 7 (a) belongs to the system description input.

In our experiment, we implement the system example on a multi-processor target architecture of three 68000 processors. Figure 7 (b) shows the result of implementation. In the figure, four Token tasks are mapped to two processors (two tasks on each processor) and the counting tasks is mapped to the other processor.

4.2. Synthesis of Application Specific OS and SW targeting

In the system description input, we assigned the information of processor mapping in the attributes of each mod-

ule. We also assigned equal priority to all the tasks. To the communication channel between modules, we specified one word communication with non-blocking write/blocking read. In the example, the size of transferred data, i.e. counter value and token, is one word.

First, the system description input is read into Architecture Analyzer. Then, Code Selector selects the following four kernel functions and services for the OS's of two processors where two Token tasks are mapped.

- Round-robin scheduler service since tasks have the same priority.
- A timer service since the round-robin scheduler is used.
- Non-blocking write (called **exoutd**) and blocking read services (called **exinb**).

Code Expander generates the OS source code that handles two tasks of equal priority and two communication service functions (**exoutd** and **exinb**). Task Code Adaptor replaces original communication functions (i.e. **outport(Port, value)** and **inport(Port, value)**) by OS communication services (i.e. **exoutd(&PortAddr, value)** and **exinb(&PortAddr, value)**).

For the processor where only the counter task is mapped, the same communication services are selected. However, no scheduler and timer services are selected since there is only one task on the processor.

We obtained three binary executables for three processors after running compilation with the generated source codes and makefiles. We validated the system implementation in cosimulation with three instruction set simulators of 68000 processor and a VHDL simulator for the HW interfaces of processors.

As a preliminary result, in this experiment, the generated OS gives very small code sizes: 797 lines in C, (in assembly) 1.86 KB for each of two processors with two Token tasks and 1.62 KB for the processor with one Counter task. In terms of performance, it gives 83 instruction cycle latency in the channel read operation from interrupt trigger to the end of single-word data access.

5. Conclusion

We proposed a method of automatic generation of application specific operating systems and automatic targeting of application code. The proposed method starts automatic generation of operating system from a very small and flexible kernel and includes only the OS services specific to the application. We applied the method to a token ring system and obtained a promising result.

References

[1] Synthetix project. available at http://www.cse.ogi.edu/DISC/projects/synthetix/.

[2] J. Cortadella and al. Task Generation and Compile-Time Scheduling for Mixed Data-control Embedded Software. *Proc. Design Automation Conf.*, pages 489–494, June 2000.

[3] D. Desmet, D. Verkest, and H. D. Man. Operating System based Software Generation for Systems-on-Chip. *Proc. Design Automation Conf.*, pages 396–401, June 2000.

[4] R. Dick, G. Lakshiminarayana, A. Raghunathan, and N. Jha. Power Analysis of Embedded Operating Systems, pp.312-315. *Proc. Design Automation Conf.*, June 2000.

[5] S. Edwards. Compiling Esterel into Sequential Code. *Proc. Design Automation Conf.*, pages 322–327, June 2000.

[6] Eonic Systems, Inc. Virtuoso v.4.1. available at http://www.eonic.com/.

[7] T. Grandpierre. Optimized Rapid Prototyping for Real-Time Embedded Heterogeneous Multiprocessor. *CODES Workshop on Hardware/Software Codesign*, May 1999.

[8] A. Österling, T. Benner, and R. Ernst. Code Generation and Context Switching for Static Scheduling of Mixed Control and Data Oriented HW/SW Systems. *Proc. 4th Asia-Pacific Conference on Hardware Description Language*, pages 131–135, Aug. 1997.

[9] B. Shackleford, M. Tasuda, E. Okushi, H. Koizumi, H. Tomiyama, and H. Yasuura. Memory-CPU Size Optimization for Embedded System Designs. *Proc. Design Automation Conf.*, pages 246–251, June 1997.

[10] F. Thoen and F. Catthoor. *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*. Kluwer Academic Publishers, Boston, 2000.

[11] S. Vercauteren, B. Lin, and H. D. Man. A Strategy for Real-Time Kernel Support in Application-Specific HW/SW Embedded Architectures. *Proc. Design Automation Conf.*, pages 678–683, 1996.

[12] Wind River Systems, Inc. VxWorks 5.4. available at http://www.wrs.com/products/html/vxwks54.html.