AMPLE: Automatic MaPping of aLgorithms for Embedded systems[1]

I. Panagopoulos, A. Dimopoulos, G. Manis and G. Papakonstantinou

Dept. of Electrical and Computer Engineering
National Technical University of Athens
Zografou 15773, Athens
GREECE
ioannis@cslab.ntua.gr http://www.cslab.ece.ntua.gr

Dept. of Computer Science
University of Ioannina
P.O. Box 1186, Ioannina 45110
GREECE
manis@cs.uoi.gr

Abstract

Presented here is AMPLE, a platform-based design methodology and its realization in a software tool for automatic mapping and performance optimization of algorithms to embedded system architectures. Source code of a high level programming language is initially analyzed through the use of a set of profiling tools. Analysis results provide insight on both the computational intensive tasks as well as information on variable usage/sharing requirements. An architectural configuration is then selected based on a typical CPU-FPGA interconnection scheme. The code is divided into parts which will execute on a microprocessor and parts which will be handled by special purpose hardware (on the FPGA of the platform). Shared memory areas defined through the use of common variables are also mapped to the memory components of the platform. Nested loops are of special interest since they are usually the most computationally intensive tasks and are automatically parallelized and implemented in hardware in the platform's FPGA. The presented methodology can effectively reduce design time and thus minimize time-to-market requirements as well as provide a considerable performance improvement in the resulting system.

Keywords: automatic algorithm mapping, embedded systems, nested loops

[1]This work is co - funded by the European Social Fund (75%) and National Resources (25%) - Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS II.

1. Introduction

The design of embedded systems from high level specifications entails conformance to two basic requirements: Minimization of the time-to-market design requirement and exploitation of the desirable trade-off between speed, area, cost, power consumption and other implementation features. The former is crucial for end-product marketing success while the latter defines the quality and efficiency of the final product. The designer's flexibility of experimenting with different architectural schemes at various levels of abstraction, migrating parts of the specification from software to dedicated hardware (on FPGAs or as ASICs) and selecting processing elements from a great variety of pre-designed off-the-shelf components forms a very large design space within which the best design trade-off needs to be explored in very strict time-to-market limits. Towards this goal, both academic and commercial tools have been presented which automate parts of the design process and offer easy experimentation and evaluation at higher levels of abstraction of different design choices. The successful application of those tools is greatly dependent on the initial information that is available to the designer concerning characteristics of the initial application's specification. The usefulness of such knowledge is two-fold. On one hand it assists the designer refine his/her initial specification and on the other hand it can lead to a more guided exploration of the design space and thus better conformance to the desired trade-off in smaller time-to-market limits.

Presented here is AMPLE (**A**utomatic **M**a**P**ping of a**L**gorithms for **E**mbedded systems). Source code of a C-like programming language is partitioned into tasks and analyzed through the use of a set of profiling tools in order to present high level information/approximations on the computational/communication intensity of each task and on the inter-task communication requirements. The source code is then divided into tasks which will execute on a CPU and tasks which will be mapped to special purpose hardware. The complexity of the code, its variables and the dependencies between them, the computational time required for the execution and the frequency of execution are the main criteria for code partitioning and characterization. Based on the communication analysis of the initial code a specific interconnection scheme is selected from a set of predefined CPU-FPGA platforms. Nested loops are of special interest since programs usually spent a lot of execution time within loops. For that reason, nested loops are not only implemented in hardware but prior to that they are automatically parallelized in special purpose architectures and then implemented in the platform’s FPGA component. The methodology/tool leads to the implementation of an embedded system by automatically compiling tasks that will be executed in Software and implementing in a synthesizable Hardware Description Language (VHDL or Verilog) parts that will be executed in hardware.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 describes the proposed system. Section 4 discusses some implementation details. The last section summarizes the features of the proposed platform and presents some plans for future research work.

2. Related Work

In mid 90's the complexity and the density of digital circuits resulted in new methods for *high level hardware synthesis*. In these methods, hardware is implemented based on specifications given in a high level of abstraction. An overview of high level hardware synthesis methodologies is provided in [Gadjski et. Al. (1994)]. In [Berkley HW/SW Codesign grp.(1994)] POLIS produces hardware, based on specifications described by finite state machines (with some extensions) while in [Fin et. Al. (2001)] the specification language is System-C, an extension to C for supporting processes and communication mechanisms between them. In [Kambe et. Al. (2001)] a tool is presented based on BachC, a subset of C enhanced with instructions and structures for hardware synthesis.

The computational complexity and the density of the digital circuits kept increasing with remarkable rates. The high level hardware synthesis methods could not cover satisfactorily the increasing demands and the time-to-market limits, mainly due to two reasons: (i) design from scratch was too expensive in terms of time and money and (ii) the high density of integrated circuits resulted in a high latency, comparable with that of transmitting signals over busses. High level hardware synthesis methodologies were partially substituted from *system level synthesis*. Now the basic structural units are more complicated modules (CPUs, memories, special purpose hardware, etc), simplifying the designers' task and reducing the cost. Typical examples of system level synthesis methodologies are [Petrot et. Al. (2001)][Rowson et. Al. (1997)][Nandi et. Al. (2001)][Martin et. Al. (2002)]. In [Petrot et. Al. (2001)] a system which produces embedded systems from SDL (Synchronous Data Language) is described and in [Martin et. Al. (2002)] another system is presented based on UML (Unified Modeling Language) this time.

Some important limitations for the system level synthesis methodologies include the complexity of the interface of the structural units which some times resulted even in the redesign of some units so that they are customized for specific applications. A more recent approach is the *platform based design*, where the designer selects from pre-designed (parameterized) platforms for embedded systems. From the behavioral description of the system, the implementation platform is selected. The source code is divided into parts which will execute on CPUs and parts which will be mapped on dedicated hardware. Two platform based designs for embedded systems are presented in [Sangiovanni-Vincentelly et. Al. (2002)] and [CoWare,INC (2004)].

The proposed methodology preserves the merits of high-level synthesis by allowing the automating generation of the architectural sub-modules, allows design diversity since it separates interface from behavior, takes into consideration memory delays in the overall performance and can be applied to any platform based on the interconnection of a CPU and an FPGA. In other words, the methodology summarizes the advantages of each design approach in a single design process.

3. *The Proposed Methodology*

In this section the architecture of AMPLE will be discussed. In Figure 1 a general overview of all design phases is shown.

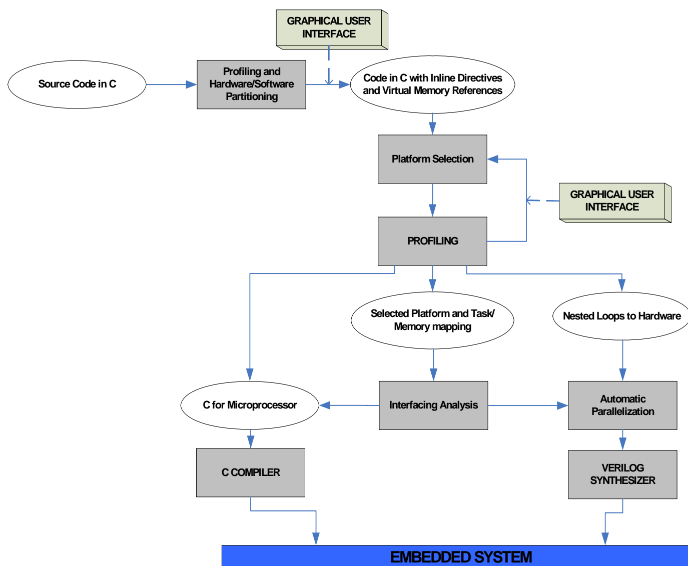

Figure 1. *The overall architecture. Rectangles represent components, ovals represent data*

The input to the platform is source code in C-like high-level programming language. The source code is fed to the hardware/software partitioning unit which (i) detects the nested loops and separates them from the rest of the code, (ii) generates static information related to the nested loops necessary for possible further separation of the code, (iii) maps all variables, which are used within loops, to a virtual memory and (iv) converts all references to these variables to the virtual memory equivalents.

The output of the hardware/software partitioning unit is presented to the user through a graphical interface. The user can evaluate the suggested segmentation and modify it if necessary. The modified code, enriched with the information associated with the

nested loops and the virtual memory representation, is used in an iterative procedure for platform selection. More specifically, from all possible ways of interconnection between the microprocessor, the FPGA, the memory or other structural elements, the most effective one is selected according to the cost, the response time, the size of the device and other general or application specific requirements of embedded systems. The user through a graphical representation compares all suggested alternative solutions and selects the one that covers the specified design requirements.

The nested loops which have been selected by the user are fed to the module "Automatic Parallelization'. In this module, algorithms for the automatic parallelization of nested loops are applied according to the loop and the dependencies of the data. The parallelized loops are described in the Verilog hardware definition language. AMPLE provides clear separation between behavior and interfacing of the implemented component. Therefore only the behavior of the parallelized implementation of the nested loop is defined. The interfacing of the hardware implementation to the rest of the platform is defined in the "Interfacing Analysis" task which ensures transparent communication of the hardware and software tasks on the platform.

The part of the code which will execute on a microprocessor is also enriched with (i) code which is responsible for the communication with the parts of the code that have been selected for hardware and (ii) code for the communication with the FPGA.

The final step is to compile the resulting Verilog code by commercial tools and produce the embedded system.

The proposed methodology presents some interesting features which differentiates it from similar platforms:

- It detects the nested loops of the source code and extracts static features used for loop parallelization. These features can give a first estimation of the efficiency of the system before completing the whole design procedure
- It allows the estimation of the communication costs and the experimentation with alternative solutions and interconnection architectures
- It ensures efficiency through innovative loop parallelizing techniques

4. Implementation Details through an Illustrative Example

In order to clearly illustrate the proposed methodology a small example will be used as the input algorithm to AMPLE. Suppose the following code is given, which is a toy scale, simplified program for simulating the effects of a constant pressure applied to two sides of a surface:

```
long [100,100] s ;
long [100,100] a ;
long [100,100] k ;
float [100,100] Pm;
long zmax;
float P,average, volume;
int i,j,side,N,time,points;
float Pall;

average=0;
volume=0;
side=5;
N=50;
points=0;

for (time=0;time<N;time++)
{
  //  calculation  for  one  simulation
circle
  for (i=1;i<100;i++)
    for (j=1;j<100;j++)
    {
     P=0.5*Pm[i-1,j]+0.5*Pm[i-2,j-2];
          if (s[i,j]<zmax)
          {
           s[i,j]+=k[i,j]*a[i,j]*P;
           Pm[i,j]=(1-a[i,j]*P);
          }
          else  Pm[i,j]=P;
    }
}
```

```
// calculation of average height
for (i=0;i<100;i++)
     for(j=0;j<100;j++)
          average+=s[i,j];
average=average/1000;

// calculation of volume
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
    volume=volume+side*side*s[i,j];

// calculation of points with marginal
height
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
    if (s[i,j]>=zmax) points++;

// calculation of total pressure
Pall=N*P;
```

4.1. Hardware/software separation

At the first step (hardware/software partitioning unit) the nested loops are detected and features concerning variables and computation intensiveness of tasks are extracted (e.g. information for the number of repetitions, the dimension of the loops, and the number of instructions contained in the loop body). The code is transformed to the following one, and the extracted information on nested loop characteristics is stored separately. The characteristics extracted for the example are shown in Table 1.

Table 1. *Loop features extracted for the pressure example*

ID	Type	Instructions	Iterations	Dimensions
2	FOR	4	10000*N	3
3	FOR	1	10000	2
5	FOR	1	10000	2
6	FOR	1	10000	2

```
//^ BLOCK:1
average=0;
volume=0;
side=5;
N=50;
points=0;
//^EBLOCK:1
//^ BLOCK:2
for (time=0;time<N;time++)
  for (i=0;i<100;i++)
    for (j=0;j<100;j++)
     {
      P=0.5*Pm[i-1,j]+0.5*Pm[i-2,j-2];
          if (s[i,j]<zmax)
           {
             s[i,j]+=k[i,j]*a[i,j]*P;
             Pm[i,j]=(1-a[i,j]*P);
           } else Pm[i,j]=P;
     }
//^EBLOCK:2
```

```
//^BLOCK:3
for (i=0;i<100;i++)
  for (j=0;j<100;j++) average+=s[i,j];
//^EBLOCK:3
//^BLOCK:4
average=average/1000;
//^EBLOCK:4
//^BLOCK:5
for (i=0;i<100;i++)
  for(j=0;j<100;j++)
    volume=volume+side*side*s[i,j];
//^EBLOCK:5
//^BLOCK:6
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
    if (s[i,j]>=zmax) points++;
//^EBLOCK:6
//^BLOCK:7
Pall=N*P;
//^EBLOCK:7
```

At the same time the variables are translated in virtual memory references as illustrated in Table 2.

Table 2.*The virtual memory for the pressure example*

Variable	Size (B)	Ele-ments	Total size (B)	Location in memory
long [100,100] s	4	10000	40000	0 – 39999
long [100,100] a	4	10000	40000	40000 – 79999
long [100,100] k	4	10000	40000	80000 – 119999
float [100,100] Pm	4	10000	40000	120000 – 159999
long zmax	4	1	4	160000 – 160003
float P	4	1	4	160004 – 160007
char i,j	1	2	2	160008 – 160009
float average	4	1	4	160010 – 160013
float volume	4	1	4	160014 – 160017
int side	2	1	2	160018 – 160019
int N	2	1	2	160020 – 160021
int time	2	1	2	160022 – 160023
int points	2	1	2	160024 – 160025
float Pall	4	1	4	160026 – 160029

The use of virtual memory is essential for providing a common addressing mechanism for all parts both in hardware and in software. The placement of variables in the virtual memory at this stage will assist later the interfacing analyzer in implementing the appropriate communication modules for read/write accesses. Moreover, the virtual memory allows an upper level examination of areas of memory that will be heavily accessed during execution and therefore make the perfect candidates for their placement at the fast memory components of the platform.

The initial automatic separation and selection of nested loops can be changed by the user through a graphical interface. Suppose that the selection of the user has lead to the following code:

```
//^TASK:1$
//DESCRIPTION:INITIALIZATION$
//MAPTO:SOFTWARE
average=0;
volume=0;
side=5;
N=50;
points=0;
//^ETASK:1
for (time=0;time<N;time++)
{
//^TASK:2$
//DESCRIPTION:SIMULATION LOOP$
//MAPTO:HARDWARE
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
  {
    P=0.5*Pm[i-1,j]+0.5*Pm[i-2,j-2];
    if (s[i,j]<zmax)
    {
          s[i,j]+=k[i,j]*a[i,j]*P;
          Pm[i,j]=(1-a[i,j]*P);
    } else  Pm[i,j]=P;
  }
//^ ETASK:2
}
//^TASK:3$
//DESCRITPION:SURFACE CALCULATIONS$
//MAPTO:HARDWARE
for (i=0;i<100;i++)
     for (j=0;j<100;j++)
               average+=s[i,j];
//^ETASK:3
//^TASK:4$
//DESCRIPTION: AVERAGE CALC$
//MAPTO:SOFTWARE
average=average/1000;
//^ETASK:4
//^TASK:5$
//DESCRIPTION:VOLUME CALCULATION$
//MAPTO:HARDWARE
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
   volume=volume+side*side*s[i,j];
//^ETASK:5
//^TASK:6$
//DESCRIPTION: MAXHEIGHT CALCULATION$
//MAPTO:HARDWARE
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
    if (s[i,j]>=zmax) points++;
//^ETASK:6
//^TASK:7$
//DESCRIPTION:     TOTAL     PRESSURE
CALCULATION$
/MAPTO:SOFTWARE
Pall=N*P;
//^ETASK:7
```

The graphical representation of the task partitioning to Hardware/Software is illustrated in Figure 2.

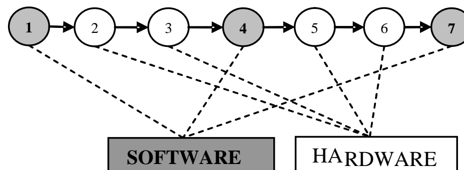

Figure 2. *Graphical representation of the task graph partitioning to Hardware/Software*

4.2 Platform Selection

The iterative procedure for platform selection assists the user in deciding amongst the following alternative architectures, also shown in Figure 3.

1. The microprocessor and the FPGA share the memory through a bus
2. The FPGA serves as a coprocessor. Only the microprocessor has access to the memory. The communication between the microprocessor and the FPGA is achieved with a shared buffer. The microprocessor controls the FPGA through control signals, launches tasks and collects the results
3. The microprocessor and the FPGA have private memory spaces and communicate to each other through a buffer
4. Similar to the second option, but the FPGA can access the memory with direct memory access (DMA technology)

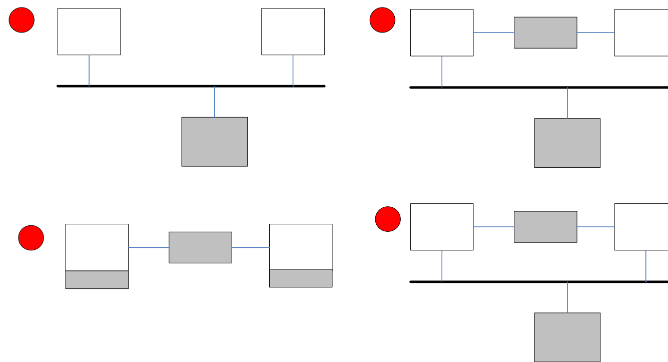

Figure 3 *Alternative available architectures for the suggested methodology*

The first selection reduces the implementation cost but increases the communication overhead. The second alternative is appropriate when the time required for the exchange of variables among CPU-FPGA is small, i.e. the amount of data which should be transferred from the microprocessor to the FPGA and vice versa is small. The third selection is appropriate for applications with tasks which do not have to communicate a lot with each other, and most of their variables are private to these tasks. The last selection has increased implementation cost, high communication overhead but combines the advantages of selections 1 and 2. A fifth solution which combines all the above characteristics is also possible, but the implementation cost is very high.

The element "Profiling" forms a mapping of the variables of all tasks to the virtual memory. In Figure 4, the mapping for the pressure example is shown and in Table 3 we illustrate the allocation of spaces according to the selected data mapping.

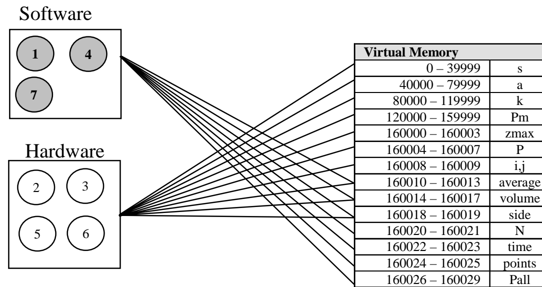

Figure 4 *Virtual memory and hardware/software mapping for the pressure example*

Table 3. *Memory access from software and hardware units*

Access	Memory space	Memory size
hardware	0 - 160003	160020 Bytes
software	160010 - 160029	20 Bytes
shared	160004 - 160025	10 Bytes

At this stage of design, the separation into software and hardware has been done, the architecture of the platform has been selected and the variables have been assigned to their real locations.

The element "Analysis of hardware/software interface" adds the necessary code for memory access. This code is selected from an available library of ready-to-use protocols. The element "Automatic parallelization" transforms loops as described in detail in [Panagopoulos (2004)]. The general architecture is given in Figure 5. The controller calculates the coordinates of the instances of the nested loop which execute in parallel (define a hyperplane). A distribution algorithm assigns these points to the buffers connecting the processing element with the coordinator. The processing elements collect the tasks their buffers, perform the tasks and then collect another task from the buffer, until no more tasks are to be performed. In other words, the controller decides about which tasks can be performed in a specific time moment and the processing elements perform the tasks. It is a dynamic load balancing technique, similar to the farmer-worker programming paradigm.

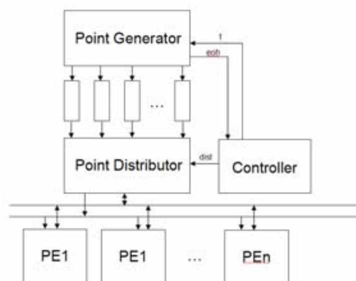

Figure 5 *General Architecture of the Automatic Parallelization Unit*

5. *Conclusions and Future Work*

AMPLE is a platform based design methodology for the design of embedded systems from high level programming languages. It comes equipped with a set of profiling tools, introduces the use of a virtual memory as a middle layer to assist memory mapping, it automatically synthesizes nested loops in HDL language and provides an analytical workspace for design space exploration. It manages to minimize design time while allowing exploitation of the most desirable performance-area tradeoff. In the future, we are planning to further expand the tool by allowing additional parallelizing methodologies to be used in the implementation of nested loops, to add further profiling information concerning dynamic features of the algorithm and to provide microprocessor specific optimization which will also affect the performance of the executed software.

References

Gadjski,D., Ramachandran,L., *Introduction to High Level Synthesis*, Proc. IEEE Design and Test of Computers, 1994, pp.44-54

Berkley HW/SW Codesign grp., A Framework for Hardware-Software Co-Design of Embedded Systems, available at: www-cad.eecs.berkeley.edu/~polis

Fin,A., Fummi,F. and Signoretto,M. *SystemC: A Homogenous Environment to Test Embedded Systems*, Proc. IEEE Codesign Conference (CODES), Copenhagen, Denmark, March 2001, pp. 17-22

Kambe,T. Yamada,A. Nishida,K. Okada,K. Ohnishi,M. Kay,A. Boca,P. Zammit,V. and Nomura,T. *A C-based Synthesis System, Bach, and its Application*, Proc. of the Asia South Pacific Design Automation Conference, 2001

Petrot,D. and Auge,I. *A Practical Tool box for System Level Communication Synthesis*, Proc. Of CODES 01, 2001, pp. 48-53

Rowson,J. and Sangiovanni-Vincentelli,A. *Interface-Based Design*, Proc. Of DAC, 1997 1997, pp. 178-183

Nandi,A. and Marculescu,R. *System-Level Power/Performance Analysis of Embedded Systems Design*, Proc of DAC 2001, 2001, pp. 599-604

Martin,G. *UML for Embedded Systems Specification and Design: Motivation and Overview*, Proceedings of the 2002 DATE 2002, 2002

Sangiovanni-Vincentelli,A. and Martin,G. *Platform-Based Design and Software Design Methodology for Embedded Systems*, IEEE Transactions on CAD of Integrated Circuits and Systems, 2002, December, Vol. 19, No 12, pp. 1523-1533

CoWare,INC *Flexible Platform-Based Design with the COWARE N2C Design System*, October, 2000

Panagopoulos,I. *Hardware/Software Codesign Techniques for Embedded Systems*, PhD Thesis, Computer Systems Laboratory, National Technical University of Athens, 2004