Contract no. 248424
FP7 STREP Project

MADNESS

Methods for predictAble Design of heterogeNeous Embedded System with adaptivity and reliability Support

D7.3: Second report on compilation toolchain

Due Date of Deliverable	**31st December, 2011**
Completion Date of Deliverable	**31st December, 2011**
Start Date of Project	**1st January, 2010 - Duration 36 Months**
Lead partner for Deliverable	**ICD**

Revision: v1.0

Project co-funded by the European Commission within the 7th Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contents

1. Introduction

The number of transistors on a chip and the frequencies used increased during the last decades. Due to the roughly scratched squared increase of energy consumption due to a linear increase in clock frequency, continuing on that path for higher computational power becomes increasingly difficult. Thus, a shift towards multiple slower clocked processors on the same chip can be observed. These processing units are often combined with additional components like memories or peripheral controllers to form so called *MultiProcessor System-on-a-Chip* (MPSoCs). For embedded systems, the trade-off between performance and energy consumption has always been at focus. Providing the demanded performance while maintaining energy constrains can often be achieved by using highly specialized processors and dedicated hardware blocks. This leads to heterogeneous MPSoCs where different processor types are combined to an energy efficient system [1].

In the MADNESS project, new methodologies for the entire MPSoC development process are proposed.

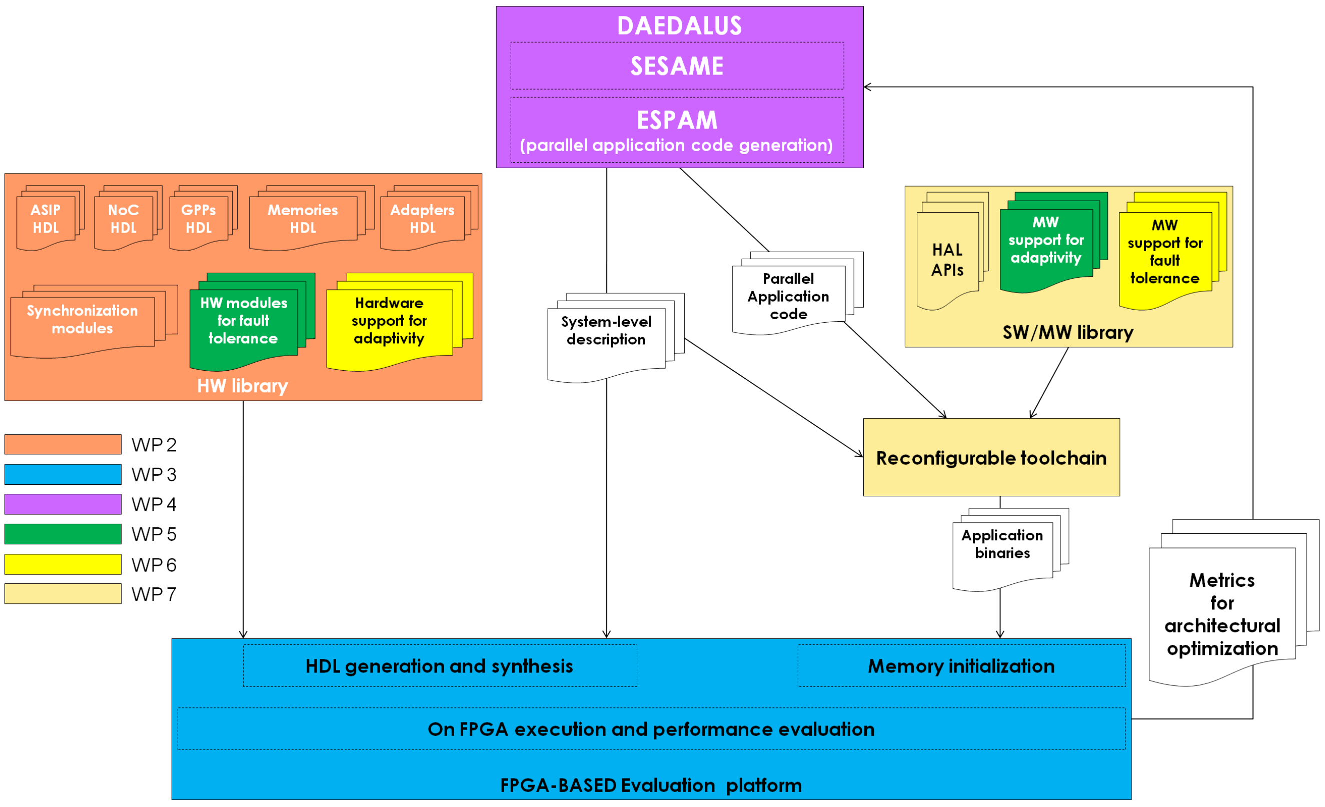
DAEDALUS
SESAME
ESPAM
(parallel application code generation)
ASIP HDL
NoC HDL
GPPs HDL
Memories HDL
Adapters HDL
Synchronization modules
HW modules for fault tolerance
Hardware support for adaptivity
HW library
System-level description
Parallel Application code
HAL APIs
MW support for adaptivity
MW support for fault tolerance
SW/MW library
Reconfigurable toolchain
Application binaries
Metrics for architectural optimization
WP 2
WP 3
WP 4
WP 5
WP 6
WP 7
HDL generation and synthesis
Memory initialization
On FPGA execution and performance evaluation
FPGA-BASED Evaluation platform

Figure 1.1: MADNESS tool flow overview

The set of tools used and developed in the MADNESS project are depicted in Figure 1.1. WP2 covers the hardware library on the left side of the figure. WP3 is responsible for the FPGA-based evaluation platform. The design space exploration tool DAEDALUS is part of WP4. The adaptivity and fault tolerance is tackled in WP5 and WP6 and distributed to the hardware library and the software library. The last WP in the figure is WP7 which takes care about the compilation and hardware abstraction. WP7 interacts directly with WP4 and WP3. Nevertheless, WP7 is important to the entire tool flow as it provides the software translation path for the MADNESS framework.

This deliverable reports on the work performed in the second reporting period. Especially, focusing on Task 7.2 where the development of a compilation toolchain and the integration of a hardware abstraction layer have been tackled. The following subsections provide a short introduction to the content of this deliverable.

1.1 MADNESS Compilation Toolchain and HAL

Beginning from either sequential or already parallelized applications specified as *Kahn process networks* (KPN) [2], an optimal MPSoC is derived from a set of predefined hardware IPs. Including the case where cross vendor IPs have to be combined like the processors from Intel or Lantiq. Thus, the translation from program's source code to the processor-dependent binary code is an important and challenging task. This compilation process is tackled in the MADNESS Compilation Toolchain developed in WP7. A hardware abstraction layer which encapsulates the access to low-level processor-dependent functionality is necessary to enable a platform-independent application development process. This abstraction layer and the definition of the provided functionality and interfaces are also part of WP7.

1.2 MADNESS demo application

The MADNESS project uses an adapted demo application to demonstrate the results of the project. All project partners agreed on a combined network and video application. The idea is to focus on an IPTV appliance, which needs extract video streams from network traffic and decoding them. This could be the case for a transcoding service on a gateway. According to real live demands and to reduce the workload on the decoder, a special preprocessing step is employed which filters the network traffic and feeds the video-decoder only with relevant packets. Thus, the video-decoder hardware does not spend effort on non-video packets. Packets which do not contain video data are forwarded to a network interface for further protocol interworking tasks.

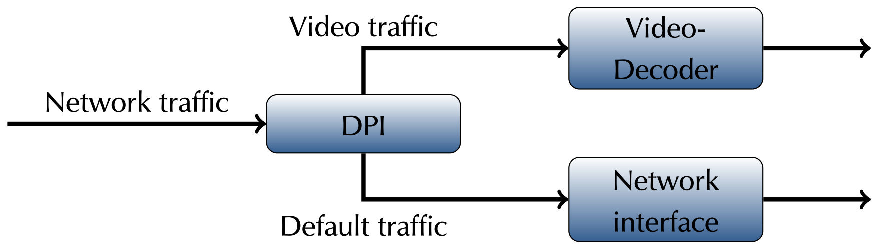

Figure 1.2: MADNESS demo application

Figure 1.2 depicts a brief overview of the MADNESS demo application. Network traffic is processed and classified, video data traffic is passed to the video decoder and other traffic is forwarded to a network interface. For classification, OpenDPI is employed and for video, H.264 is used. For the work performed in WP7, the deep packet inspection code will be used as a demonstrator.

1.2.1 OpenDPI - A driving example

To demonstrate the techniques developed in WP7, OpenDPI is chosen as an example application. OpenDPI is available as open-source, in MADNESS version 1.2 is used [3]. This library is designed to detect high-level protocols within traffic flows in networks.. Thus, OpenDPI can be used to classify network packets and for example to provide Quality of Service (QoS) mechanisms. To classify packets, deep packet inspection techniques are utilized. To get a better understanding how OpenDPI works, a more detailed description is given.

In the original Linux based example shipped with OpenDPI, the application utilizes *libpcap* [4] to read packets either from a file or a real physical network interface. Each packet is processed sequentially in a loop and passed to the detection engine. There, the packet's sender and receiver information is extracted. Afterwards, OpenDPI checks the protocol type (i.e. Transmission Control Protocol (TCP) or User Datagram Protocol (UDP)). This information can be extracted from packet's IP header. Analogously, the analyzer goes deeper into the packet structure and uses bitmasks to detect used protocols in packet's payload. Finally, the program displays the results of the analysis to the user. A modified version of this example program is used in MADNESS.

Until now, the example application was designed to work on a Linux PC system only. In addition to the assumption on the operating system, special assumptions on the underlying hardware were made. Especially a specific memory layout or size of data types were implicitly assumed. Furthermore, applications in MADNESS are specified as process networks. Thus, modifications to the original example code were necessary. In the following, these modifications are described in detail.

MADNESS OpenDPI modifications

To use the OpenDPI example in the MADNESS project, the following modifications were necessary. First, the whole application needs to be specified as a Kahn Process Network. To exploit the inherent parallelism, the tightly coupled detection loop was split up and distributed over different processing nodes which are communicating through FIFO channels. In a first step, the detection engine has been preserved as sequential block but the preprocessing and post processing are distributed and running in parallel to the deep packet inspection algorithm. The preprocessing step receives the low-level package-stream from the network interface, preprocesses it to fit the following step and pass the data to the succeeding node. Due to the Linux related implementation, the original source used *libpcap* for low-level packet-stream acquisition. Here further adaptation was required to remove the dependency on this library. Still, the overall processing scheme had to be preserved. The detection node waits for input data from the preprocessing step; if data is available the original detection step is performed. According to the detected protocol, the packet is passed to the corresponding output FIFO. More information on actual implementation can be found in section 2.3.

1.3 Conclusion

Recent embedded systems use multiple processing units (MPSoC) to raise their performance. In addition, often these system contain different highly specialized units to achieve additional performance raise. In MADNESS according to a application specification represented as a Kahn process network, a design space algorithm generates a suitable hardware platform. For each iteration of the design space exploration the application code needs to be compiled in a different way. The MADNESS Compilation Toolchain handles this task in the context of this project. To provide the required performance, utilizing processor-dependent features within the application code is unavoidable. To provides the flexibility required for assigning parts of the application code to different processing unit, a hardware abstraction has to be used. The MADNESS demo application has been adapted to stick to that paradigm. It serves as a demonstrator for the entire MADNESS process. In this Workpackage the network classification part based on the OpenDPI example is used to show the results of this Workpackage.

2. Compilation Toolchain and Hardware Abstraction

Compiling high-level source code of an application to a low-level binary executable is done in the compilation toolchain. Compiling code for heterogeneous MPSoCs is quite tedious, even more if processors from different parties are involved. Thus, a flexible, extensible and configurable compilation toolchain is necessary to generate executables for different platforms.

Testing software for prototyping in simulation environments is a common technique to reduce design time. In MADNESS, in addition to the simulation environment, an emulation environment is developed in WP3 and used to speed up the performance evaluation during the design space exploration process. Nevertheless, a configurable multiprocessor simulation environment is useful during the design of applications for multiprocessing environments.

The task of developing applications for MPSoCs has some contradicting demands, where finding a balanced solution is a quite tedious task. An often observed example is the exploitation of processor-dependent features. On the one side, exploiting such features will increase the efficiency of the application code, or may even be required for meeting time constrains. On the other side, the requirement for highly portable code which may eventually be often reused will require a as generic as possible implementation. Thus, a way to cope with these demands is to introduce a flexible hardware abstraction mechanism and management layer. Such an layer needs to support integration of new processors or components into the application development process, ideally without modification of the application code. Furthermore, any overhead resulting from portable application development must be avoided.

2.1 Compilation Toolchain

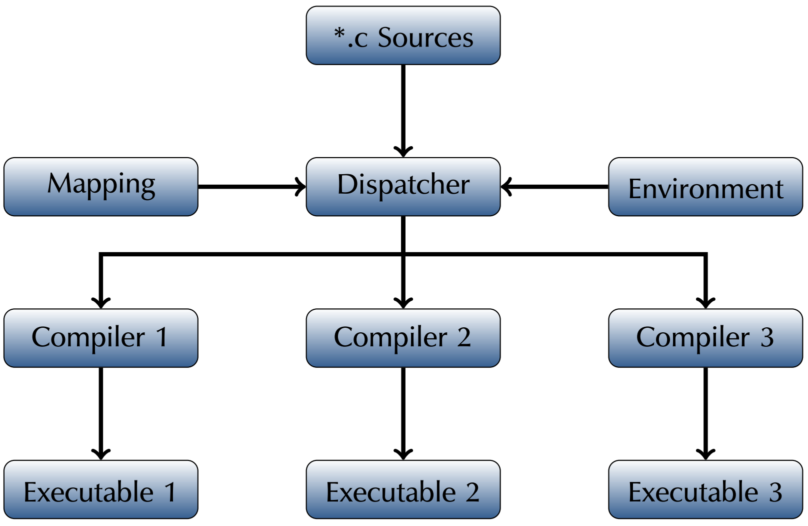

Figure 2.1: MADNESS Compilation Toolchain overview

The general idea of the compilation toolchain was already described in the first deliverable of WP7. Thus, just a short overview of the functionality is presented next. As depicted in Figure 2.1 the application source code is passed to the dispatcher. Additionally, along with the application code a mapping description which specifies which code part will be executed on which processor and an environment specification are passed to the dispatcher. The environment specification defines the available processors and their compilers. Within that file also required libraries, files to be included and compiler options are specified. The dispatcher calls the corresponding compilation tools corresponding to this description to generate the executables for each processor.

Listing 2.1 shows an environment specification where two protocol processors are involved. The `<environment>` tag defines a new environment with name *env1*. In the next line, a compiler with specific options is defined. Inside the compiler definition, paths are set to the necessary files for the compilation process. Finally, the processors for which this compiler definition should be used is assigned.

Listing 2.1: Environment specification example for the MADNESS compilation toolchain using Lantiq's compiler suite.

```
<environment name="env1">
  <compiler name="pp32cc" options="-O3" include-prefix="-I" library-prefix="-l"
output-prefix="-o">
    <!-- global source path -->
    <source-path >examples/</source-path> <!-- path where process files are
stored -->
    <!-- global include path -->
    <include-path >examples/</include-path>
    <!-- global library file -->
    <library-file >/RUNTIME/libruntime.o</library-file>
    <!-- global output path - if empty outputpath=sourcepath -->
    <output-path>/examples/out/</output-path>

    <processor name="PP32V2_1" options="--16bit" \>

    <processor name="PP32V2_2" options="--shift-arithmetic">
      <library-file>RUNTIME/libruntime_special.o </library-file>
    </processor>
   </compiler>
</environment>
```

In Listing 2.2 one can see how an environment discription for a Intel processor using the hiveCC compiler looks. In this case, a special platform description is necessary to be passed to the compiler. Thus, it can generate the correct code for the used processors.

Listing 2.2: Environment specification example for the MADNESS compilation toolchain using hiveCC.

```
<environment name="env2">
  <compiler name="hivecc" options="-m madness_platform -D__HIVECC -D__madness_platform"
      include-prefix="-I" library-prefix="-l" output-prefix="-o">
    <!-- global source path -->
    <source-path >examples/</source-path> <!-- path where process files are
stored -->
    <!-- global include path -->
    <include-path >examples/</include-path>
    <!-- global library file -->
    <library-file >/RUNTIME/libruntime.o</library-file>
    <!-- global output path - if empty outputpath=sourcepath -->
    <output-path>/examples/out/</output-path>
```

```
      <processor name="MADNESS_PLATFORM" >
        <library-file>RUNTIME/libruntime_special.o </library-file>
      </processor>
    </compiler>
  </environment>
```

With the environment specification, different compilers or compiler configurations can be used without any changes to the existing toolchains. Furthermore, new compiler and processors can be added easily. Thus, a fast prototyping and automatic compilation during the design space exploration is possible.

2.1.1 MPPSIM

The development of applications for MPSoCs is a complex task. The developer needs to take care of concurrency and synchronization of different tasks. Thus, tools which help the developer in the development of multi-task applications are in demand. Multiprocessor simulators are such helpful tools. For fast prototyping and application development, a multiprocessor simulator for Lantiq's Protocol Processor (MPPSIM) was developed. MPPSIM is based on the original single-core simulator where some extensions are made to support multiple processor cores. Furthermore, the memory subsystem simulation has be reworked. Configurable processor-local and global memories are added. Theoretically, an unlimited number of cores and memory combinations can be composed to a multiprocessing environment just limited by the performance of the host system which runs the simulator. With the enhanced simulator, the distributed deep packet inspection demo example, described in 1.2.1, was developed and tested successfully.

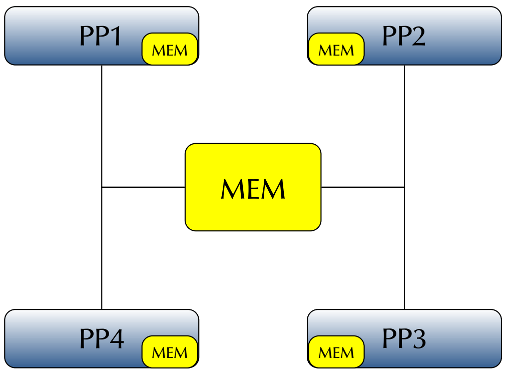

Figure 2.2: Example of a four processor configuration with multi protocol processor simulator(MPPSIM)

Figure 2.2 shows an example simulator configuration. Here, four Protocol Processors with local memories are instantiated and connected to a shared-memory. Since no real interconnect implementation is simulated, results are only useful for testing and prototyping purposes of applications.

Following, all key features of the MPPSIM, developed in this WP, are summarized:

- Configurable number of PP cores
- Configurable size of local processor memory
- Configurable size of shared-memory
- Global clock
- Cycle-true simulation

2.2 Hardware Abstraction Layer Integration

Processors used in the embedded system domain usually are designed for special purposes (i.e. video decoding). Thus, to take full advantage of such specialized processors, application specific instructions have to be used in the application code. Highly specialized compilers are able to use such instructions. Otherwise, intrinsics or compiler known functions are provided and the application developer needs to take care of using these mechanisms. An example is the RDRAND intrinsics for random number generation in Intel processors [5]. Sometimes, the only feasible solution to use of such instructions is the usage of low-level assembly instructions embedded in high-level source-code. Vector processors perform best if the assembly instructions are written in a specific order to utilize the available slots in the vector processor pipeline. Thus, scheduling of the instructions to utilize processor's vector pipeline is necessary to optimize the performance of the executed code. This explicitly scheduled code may lead to disadvantages on different processors which can be avoided by using abstraction mechanisms.

In heterogeneous systems, manual code optimization for each processor is time consuming and error prone. During design space exploration like in MADNESS, the underlying hardware platform is continuously changing. Thus, optimizing or even using processor-dependent features is almost impossible. This leads to low utilization of deteriorated the benefits of multiprocessing. Therefore, a hardware abstraction layer which makes processor-dependent features with standardized interfaces accessible is necessary.

Hardware abstraction is the key feature in developing portable applications which can be deployed on several hardware platforms. Good hardware abstraction easily allows to create application for different platforms. In MADNESS, during the design space exploration phase, platform independent applications implementations are mandatory. Within this approach, abstracting functions are classified into standard-functions and special-functions [6]. Managing hardware abstraction functionally is done in the MADNESS Hardware Abstraction Layer Data Base (HAL-DB). Nevertheless, with the tight integration of the database and the MADNESS Compilation Toolchain, correct implementations for these functions are selected automatically for each processor. In the following section, examples for different usage of the abstraction layer are given.

2.3 Examples

The enhanced OpenDPI example is used to demonstrate how the MADNESS hardware abstraction layer can be used to implement different communication methods for efficient data transport between nodes. A second example shows how an application can be deployed without changes on different platforms. The last example demonstrates the capability of the abstraction layer to provide different processor-dependent implementations for a given algorithm.

2.3.1 Different Communication Mechanisms

The first example demonstrates different communication mechanisms. In Kahn process networks, nodes are communicating through channels which are realized as FIFOs. Thus, writing to channels is in general non-blocking and reading blocks if no data is available. For a HAL implementation the read and write functions are mandatory and available for each processor used in MADNESS. The implementation of these functions is not fixed. For example, for some platforms hardware based FIFOs are used to implement the communication. Otherwise, software implemented circular buffers can be used as FIFOs. Thus, it does not matter how communication is implemented as long as the read and write interfaces provided stay the same.

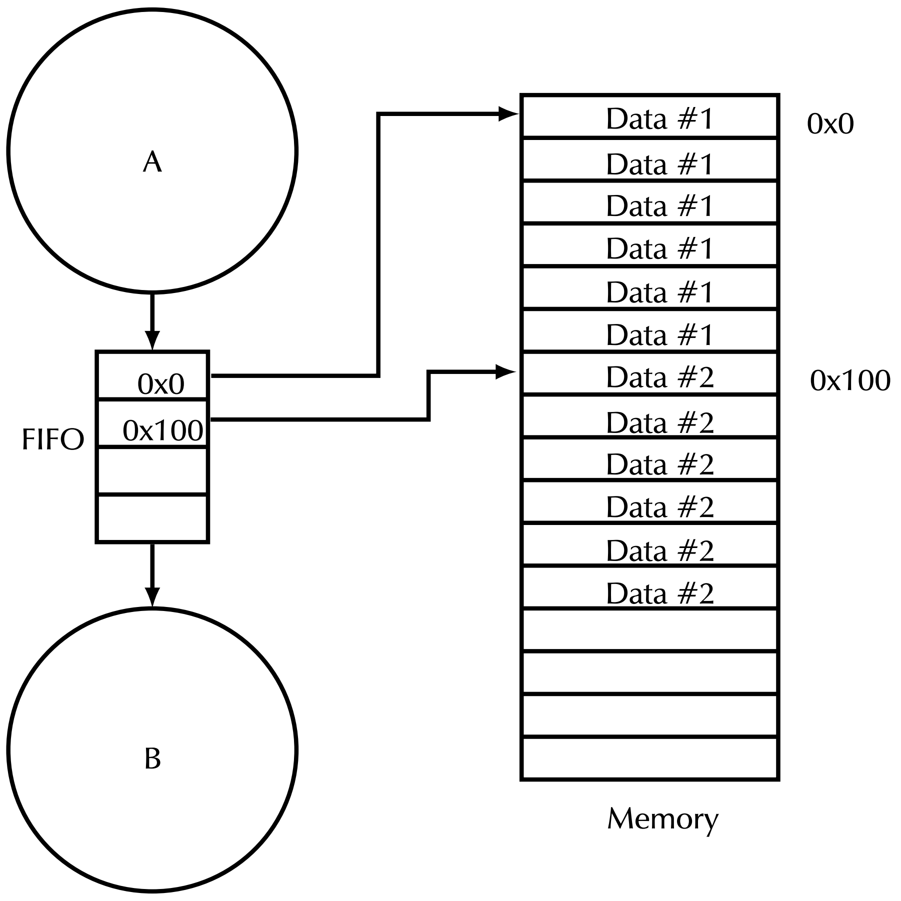

Figure 2.3: Address FIFO

Data-FIFO

The so called data-FIFO communicates the whole date structure at once. For the OpenDPI example, the entire network packet is transfered to the FIFO. Differently from other applications, OpenDPI employs large data structures which can extend to the size of whole packets, making unfeasible to use data-FIFOs. With data-FIFOs, no further synchronization between read and write accesses is necessary.

Addr-FIFO

To tackle the above-mentioned issue with huge data structures' overhead, in packet processing it is common to store packet data only once to avoid overhead.In addition, to classify packets, time constrains need to be considered and met. Thus, copying large network packets from one node to another node leads to non acceptable timing violations.

A solution is to communicate addresses of memory locations in the global shared-memory between nodes only. Thus, only a few copy operations are necessary to move a small amount of data between sender and receiver nodes. With this mechanisms the relative order of data, arriving at the receiver, is preserved. Nevertheless, special attention to memory management is important to keep the communicated data valid as long as needed. The entire management process is encapsulated in the read and write primitives provided by the MADNESS-HAL. Figure 2.3 gives an overview of the functionality of address-FIFOs.

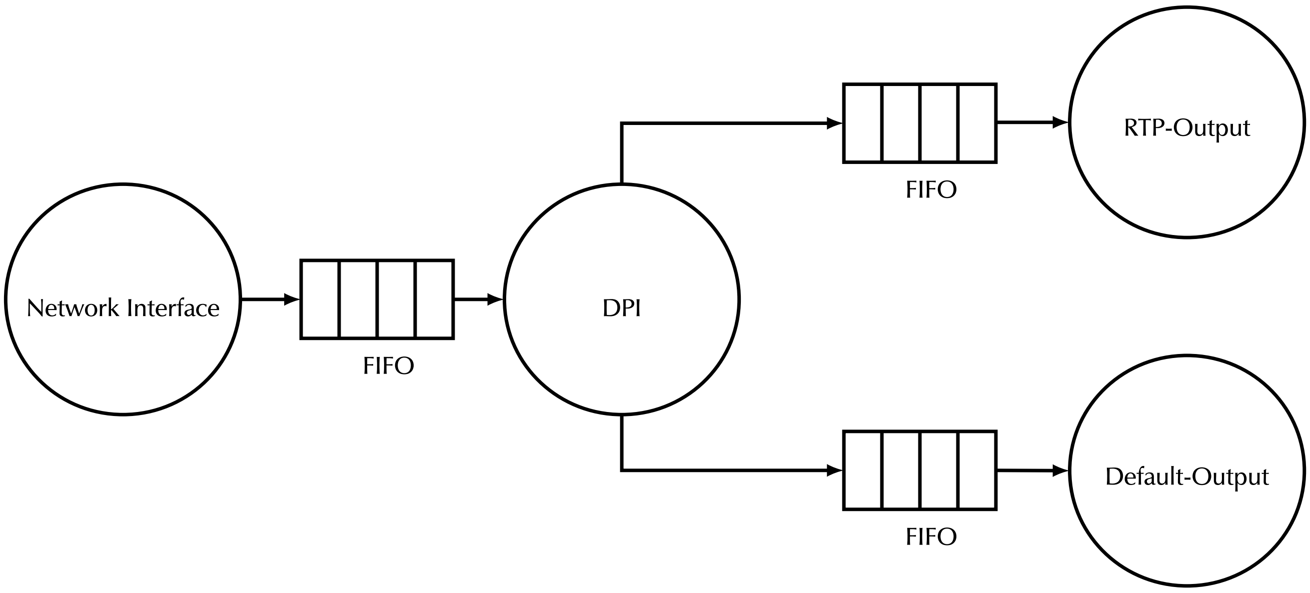

Figure 2.4: Kahn process network of the deep packet inspection example simulated with MPPSIM.

2.3.2 Different Platforms

The second example targets the development process of platform-independent multi threaded applications. During the development it is common practice to execute the program on different platforms to evaluate the application behavior. To enable application tests on different platforms a tool was developed which allows the execution of Kahn process networks on a Linux operating system. The tool implements nodes as POSIX threads [7] and manages FIFO communication with a software base a circular-buffer implementation for each channel. In combination with the standard MADNESS-HAL communication primitives, a portable application can be easily created. Thus, developers can test application's semantic on a standard computer in a straight forward way.

Figure 2.4 depicts a KPN version of the MADNESS modified OpenDPI example which is deployed on the MPPSIM described in section 2.1.1. The same application can be executed on a Linux system without modification which is shown in Figure 2.5. The grey box shows how the KPN is embedded in a Linux pthread environment.

2.3.3 Low-Level Functionality

The utilization of the full power of a specific processor requires often the usage of special low-level instructions, especially, once memory layout issues pop up. Unfortunately a common practice, which has also been observed in the OpenDPI code is to implicitly rely on the byte ordering or memory organization of the target platform. Here using processor specific instructions is often required to avoid such dependency. Assume a theoretical processor which has special instructions for efficient memory access. This processor uses a different memory layout, thus memory offset computations and data masking needs to be modified. For OpenDPI in particular, the access to header structures for network packets was affected by this issue. In contrast to modify each access, in MADNESS the physical layout of the header is modified to be independent of the memory layout and that way meet the requirements of the processor. After processing, the original memory layout is restored. With the hardware abstraction layer a function was developed which patches the header. For the highly specialized processor assumed here a processor-dependent implementation is developed which utilizes the efficient memory access low-level instructions embedded in high-level code.

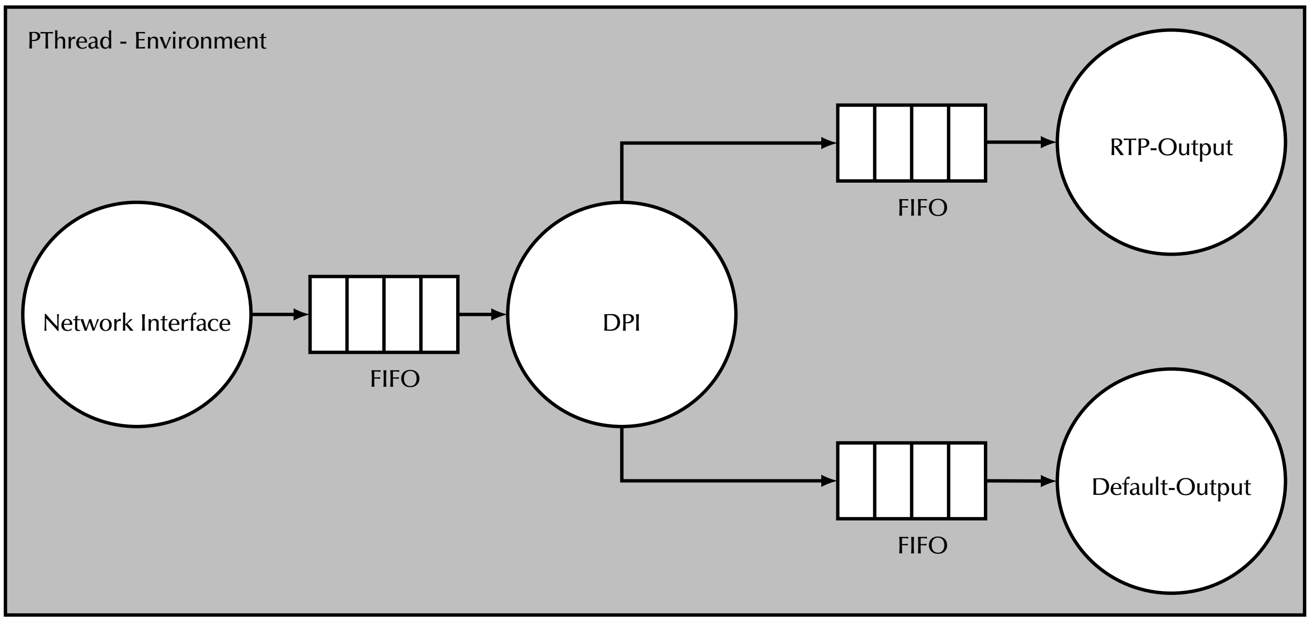

Figure 2.5: Kahn process network of the deep packet inspection example in a pthread simulation environment.

For more detailed information, please refer to deliverable 7.2 of this work package.

2.4 Conclusion

Within this section the MADNESS Compilation Toolchain has been described. The compilation toolchain provides a configurable approach to direct the application compilation process for a heterogeneous MPSoC from within a single interface. Selecting and configuring the target dependent tools and assigning the right application code parts are the primary tasks for the compilation toolchain.

For the application code, the need for an hardware abstraction layer has been identified. Several classes on abstraction functions are provided, some mandatory, mostly dealing with communication, some target special which focus on exploitation of advances features of a particular processor and finally some, which are application related, providing encapsulation of dependencies on particular target platform properties like memory layout or addressing modes. With respect to the HAL, the compilation toolchain provides methods to select the suitable implementation for each processing unit.

3. High-level Compiler Supported overhead-free Function Handling

Hardware abstraction is usually realized with C functions where for example the MADNESS Compilation Toolchain decides which low-level implementation should be used. In embedded systems, the computing performance is limited and huge function call overhead must be avoided. This requires efficient techniques to realize hardware abstraction.

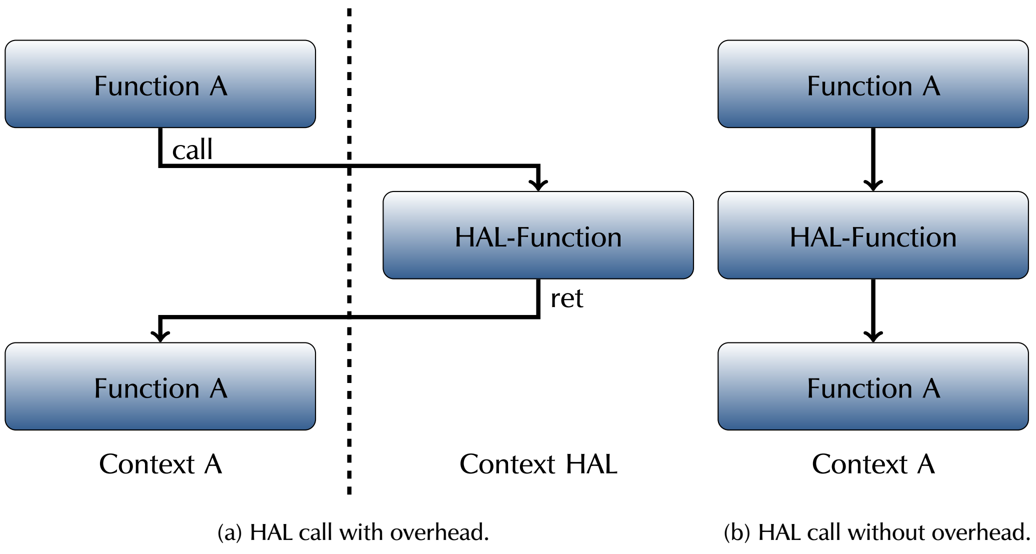

Figure 3.1: Hardware abstraction layer calls

Figure 3.1a visualizes the overhead caused by calling a function. Calling a function is coupled with changing the execution context. Thus, by executing a hardware abstraction function, the context (i.e. registers), needs to be saved and restored after returning from the called function. If this happens at a very fain grained and therefore frequent level, it is not acceptable for resource-restricted systems, which is typical for embedded environments.

In resource-restricted environments or systems where additional calling overhead must be avoided, function calls without overhead are desirable. Figure 3.1b depicts an overhead free hardware abstraction. In the following, common techniques to reduce or even completely avoid overhead are discussed. Afterwards, a new technique is presented which utilizes a compiler development framework for source code analysis and translation.

3.1 Current Solutions

Currently two major approaches are state of the art, once it comes to provide overhead free function calls as required for an efficient hardware abstraction layer. For both approaches, advantages and limitations are explained next.

3.1.1 Macros

The C preprocessor is capable of realizing hardware abstraction without overhead. The preprocessor is a separate tool and usually the first step in the compilation process. Macros are very popular for definition of constants or small function like behavior. The developer specifies keywords which will be expanded to arbitrary characters during the preprocessing step. Listing 3.1 defines in the first line a macro substitution, more precisely a macro function which is a macro with arguments. The `add(a,b)` function can be used in the following source code like a regular function. In Listing 3.2 the call to the add macros has been expanded. With this technique the developer has the benefits of a function like syntax but without overhead of actually calling a function in a new context [8].

Listing 3.1: Utilize macro functions.

```
#define add(a, b) (a+b)

int main(){
  int result = add(4, 2);
  return result;
}
```

Listing 3.2: Result after preprocessing of macro function from Listing 3.2.

```
int main()
{
  int result = 4 + 2; // Valid
  return result;
}
```

The preprocessor does not check if the arguments of such an macros are valid. Listing 3.3 shows a invalid use of the previously mentioned macro `add(a, b)`. Here, a string argument is passed which will be expanded to `4.5 + "2"`. In C language it is not valid to add a string to a floating number. Thus, the compiler will cause an error and stop compilation with a more or less confusing error message because the information, whether this statement comes from a macro or a regular line of code, is lost. This makes debugging more complex.

Listing 3.3: Wrong use of macro functions arguments.

```
#define add(a, b) (a+b)
int main(){
  int result = add(4.5, "2");
  return result;
}
```

Listing 3.4: Macro function from Listing 3.4 expanded.

```
int main(){
  int result = (4.5 + "2"); // Error
  return result;
}
```

Listing 3.5 shows another major problem with preprocessor macros taken from [8]. Here, both passed arguments are evaluated twice which leads to a unexpected behavior of the program. The expanded version of this example is shown in Listing 3.6. It is obvious that either `i` or `j` will be increased two times depending of the result of the conditional statement which is usually not desired by the developer nor the user of the macro.

Listing 3.5: Double evaluation of macro arguments [8].

```
#define max(A, B) ((A) > (B) ? (A) : (B))

max(j++, i++) /* WRONG */
```

Listing 3.6: Double evaluation of macro arguments expanded from 3.5

```
((j++) > (i++) ? (j++) : (i++))
```

3.1.2 ANSI-C inline

With C99[9], the `inline` function specifier was introduced to the standard. The specifier gives a hint to the compiler to execute the function as fast as possible [10]. Inline expansion of functions is just an optimization and the compiler has full control of which functions are going to be expanded and which are called in the traditional way. Nevertheless, functions with `inline` specifiers are passed through the same analysis phases in the compiler as normal functions. Thus, errors are precise and the issue can be tracked and solved easily.

In contrast to macros where the developer is urged due to the syntax to pass only small code fragments, inline functions are syntactically regular functions of arbitrary size. Therefore, using inline functions tends to lead to bigger code size and thus higher memory demand. This demonstrates how tightly coupled speed and code size is. Every time a function is expanded, the code size raises of the size of the function. Thus, smaller functions are usually better candidates for inlining [11] .

Listing 3.7: ANSI C99 function inlining.

```
inline int add(int a, int b){
  return a+b;
}

int main(){
  int result = add(4,2);
  return result;
}
```

Listing 3.8: ICD-C ANSI C99 function inlining and further optimizations with the tool *irinfo*. Requires optimizing level O3 and is highly compiler-dependent

```
inline int add(int a, int b){
  return a + b;
}

int main(){
  return 6;
}
```

In Listing 3.7 the function `add(int a, int b)` takes two integer arguments and returns an integer. This function is also specified to be inlined which is a hint to a compiler. If the compiler decides to expand the function inline, the result looks like listing 3.8. The `add(...)` function call has been inlined and with additional compiler optimizations evaluated to `6`. Since compilation is done in translation units a compiler is not able to detect if such a function definition is needed in other translation unit. Thus, the original function definition is not deleted and may wast memory space if the function is not needed anymore.

Nevertheless, the `inline` function specifier provides a standardized method to give hints to the compiler which functions should be inline expanded. The compiler decides which function call should be expanded. Thus, it is possible that a function with two calls will be only expanded in one case and the developer has no control and only limited insight how the compiler proceeds. The C compiler of the GNU Compiler Collection (gcc) [12] introduced a special keyword `__attribute__((always_inline))` to force the compiler to always inline code. The `__attribute__` is not a part of the original C99 standard and therefore a highly compiler-dependent extension.

The definition of the function to be inlined needs to be visible in the translation unit for the compiler, thus, functions with inline specifier are limited to one translation unit. If the same function is used in different translation units, an error occur during the linking of the resulting translation units due to multiple definitions of the same function. To overcome this issue, static inline functions can be used which limits the function scope to a single translation unit. But if the implementation of a static inline function is moved to a separate header file which allows easy maintenance of the source code, each translation unit contains an implementation of the function. During the linking process, each of these functions will be linked to the executable. This raises the size of the resulting executable and may diminish the benefits of the inline expansion.

3.2 ICD-C User Functions

Available solutions discussed before have disadvantages which are not acceptable in the MADNESS project. On the one side, preprocessor C macros are very popular and realize full function inline expansion but are hard to debug and do not provide any syntax or type checks for macro arguments. On the other side, the standardized inline function specifier is popular as well but only gives hints to the compiler and thus gives the developer no control over the inlining mechanism. Additionally, inline expansion is limited to one translation unit. The technique developed in the MADNESS project overcomes the disadvantages of the previous presented solutions.

During the MADNESS project, a new high-level source-to-source optimization plug-in was developed. This optimization allows the developer to annotate function which must be inline expanded. With the special keyword `_Pragma("CKF_USER_FUNCTION")` the developer controls this new optimization. The `_Pragma` operator is part of the C99 standard to add new functionality.

The implementation is based on the ICD-C compiler development and source level transformation framework. Of particular importance to this approach is the source level based handling of application code in ICD-C. It provides parsing of the source code and performing High-Level optimizations. As an additional benefit, in ICD-C, all high-level information except formating and comments is preserved. These features make ICD-C the optimal choice for the implementation of Source-to-Source optimizations as required for this approach. [13].

In the following example the results of this optimization are shown.

Listing 3.9: ICD-C USER DEFINE function inlining.

```
int _Pragma("CKF_USER_FUNCTION") add(int a, int b){
  return a+b;
}

int main(){
  int result = add(4,2);
  return result;
}
```

Listing 3.9 shows the annotated function `add(...)` which adds two integer values. This function is called inside the main routine. After passing this code to the inlining optimization technique without further code optimization enabled, an application code as shown in Listing 3.10 will be generated. Variables are renamed and the function is expanded. Additionally, the original function definition is removed to reduce code size. Within this simple example it is obvious that the semantics of the original code has been preserved. If additional ICD-C optimizations are enabled, the optimal result is listed in Listing 3.11.

Listing 3.10: ICD-C compiler kown user function inlining. With no optimization enabled. The ICD-C compiler framework renames variables to unique identifies.

```
int main()
{
  int __5;  int __6;  int __7;
  {
    __6=4;
    __7=2;
    {
      __5=__6 + __7;
      goto __4;
    }
   __4:
    ;
  }
  int result=__5;
  return result;
}
```

Listing 3.11: ICD-C compiler kown user function inlining. With default optimization O2 enabled

```
int main()
{
  return 6;
}
```

To inline function calls detailed knowledge about the program is necessary. The ICD-C framework provides several analysis approaches to extract information from the source code. The two important analysis results needed for function inline expansion are data dependencies and the call graph. The data dependency describes where a variable is used and defined. This information is necessary to embed function arguments in the caller context. If the caller context and the called context contain two variables with the same name `a`, with data dependency analysis connections between those variables can be identified. If no connections between the variables exist, renaming of at least one variable is necessary.

With a call graph, the calling order of a program is extracted. This information is necessary to detect calls to functions which should be inlined and to detect recursive calls. A recursive call is a execution path where only functions annotated with `_PRAGMA("CKF_USER_FUNCTION")` are executed. Figure 3.2 depicts such recursion. The functions `a()` calls `b()` which calls `c()` which calls `a()` which is a cycle where only inline functions are visited. These circular calls can not be inlined since they would result in an infinite call expansion. If at least one call on a cycle is not annotated with the inline pragma, the inline expansion is possible and is performed through methods provided by the ICD-C framework. A new analysis is implemented and added to the compiler framework to detect recursive inline calls in arbitrary source code.

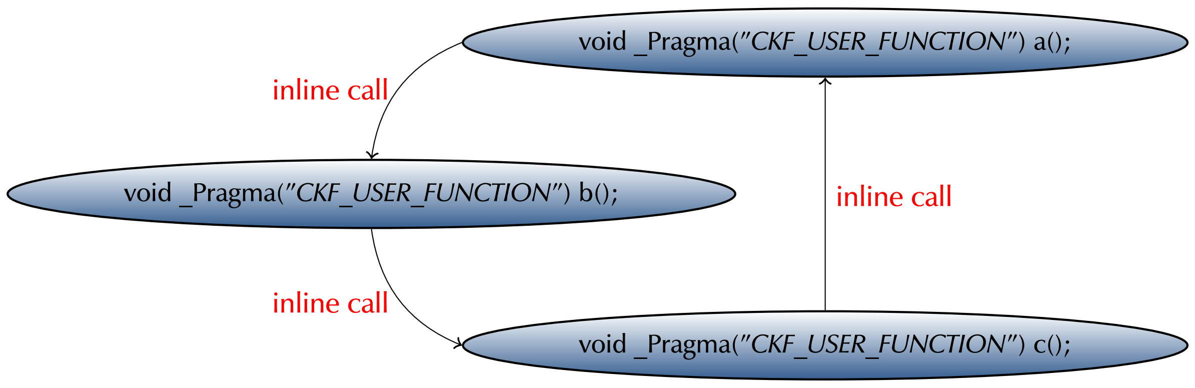
void _Pragma("CKF_USER_FUNCTION") a();
inline call
void _Pragma("CKF_USER_FUNCTION") b();
inline call
inline call
void _Pragma("CKF_USER_FUNCTION") c();

Figure 3.2: Recursive call of inline functions

3.3 Conclusion

Table 3.1 summarizes the properties of approaches discussed in this section. Different mechanisms were analyzed and evaluated in terms of hardware abstraction. Preprocessor C macros are a popular technique to realize full overhead free function calls but macros have usability disadvantages. With the introduction of the `inline` function specifier, a powerful standardized mechanism was introduced in C99 to realize inline function expansion. Nevertheless, `inline` gives only a hint to the compiler which can freely decide where a function should be expanded. Further, this mechanism may waste code size with unused function implementations. A new mechanism was developed which utilizes the power of the ICD-C high-level compiler framework to realize a overhead free hardware abstraction layer implementation combining the advantages of both approaches. Thus, functions are inline expanded anytime a annotated function is called which gives the developer the full control of a powerful inline technique. In addition, unused function definitions are deleted to reduce the code size. Further, full ICD-C capabilities are used to check the correctness of the arguments of the function to be inlined. As this optimization is implemented in a source-to-source way, the optimization can be used in several scenarios with an additional preprocessing step.

	ANSI-C	Macro	CKF_USER_FUNCTION
Advantages	standardized (C99) popular	popular true inline	true inline type-safe full control source-source optimization
Disadvantages	limited to one translation unit (unused) function definition not deleted no control	hard to debug not type-safe	additional preprocessing step

Table 3.1: Comparison of inline expansion feature's properties for C99, preprocessor macros and CKF_USER_FUNCTION

4. Conclusion

This second report on the compilation toolchain concludes the work at the heart of the software processing steps within the MADNESS project. Task 2 in Workpackage 7 is expected to provide a compilation toolchain which can be used in the synthesis process to generate the binary executables for current platform configuration.

The work performed within this task can be divided into two major aspects: on one side the development and implementation of the integrated compilation toolchain as described in Section 2, on the other side additional work was required to provide an overhead free support for hardware abstraction and other libraries requiring highly efficient implementations. Especially in the targeted context of embedded systems with their tight timing constrains and limited energy budgets, efficient exploitation of the hardware platform is of utmost importance. Section 3 presents the work performed in this subtask.

With respect to the MADNESS Compilation Toolchain the targeted results were achieved. Within the MADNESS framework the application software can be represented in a platform independent way. Due to a hardware abstraction library the application code may be compiled without manual modification for several processing unit types. The compilation toolchain accepts an environment specification and the application code in a multi threaded representation. According to the environment specification the corresponding processing unit dependent compilation tools are invoked. The compilation toolchain sticks to the interface definitions as depicted in the overall project framework overview. Therefore, the exploitation in the context of such a framework based design space exploration is feasible.

Since the compilation toolchain resides on a central location in the software definition path within the MADNESS framework, effort was required on several interfaces. First of all, the environment description is basically the interface towards the design space exploration framework. Here a XML based representation was chosen, enabling fine grained control of the underlying target dependent tools. Design as well as implementation effort was spent, enabling the successful invocation of platform dependent compilers on the relevant code fragments according to this description. Towards the application code another interface exists, since abstraction of hardware properties is mandatory in the context of target independent compilation and design space exploration. The application code exploits the features of the HAL as developed within the context of WP7. Within that context the compilation toolchain has to choose the right HAL implementation, and provide the target compiler with suitable libraries. Here effort was required in the definition process of the HAL interface, implementation of HAL libraries for MADNESS relevant processor types (i.e. Lantiq network processor) as well as for providing a generic Linux-Host based implementation for fast software evaluation and development. Finally the compilation toolchain provides an interface towards the execution environment.

Besides the immediate contribution to the workpackage result according to the DoW, additional effort was required to successfully accomplish the work for this task. First of all, effort has been spent in adapting and developing a test and demonstration application for the compilation toolchain and the corresponding HAL. Here the OpenDPI packet inspection code has been chosen. Several modification were required to adapt the code for HAL usage and fitness for a multi core environment. Within that context, extension to the simulation environment for the Lantiq network processors was performed. Furthermore, in the course of defining the HAL several limitations with respect to providing overhead free library functions of current compiler techniques have been encountered. Therefore, a source-level based transformation technique exploiting user controlled inlining has been developed. Within the context of the MADNESS HAL, the successful integration of this technique into the compilation process can be concluded. Not limited to, but especially in the context of the Lantiq processor, this transformation technique enables a highly efficient HAL implementation.

Providing a functional compilation toolchain and integration of the hardware abstraction layer support within that one, we can conclude having achieved the required results for Task 2 in Workpackage 7 for the second reporting period according to the MADNESS DoW.

Bibliography

[1] P. Marwedel, *Embedded Systems Design - Embedded Systems Foundations of Cyber-Physical Systems*, 2nd ed. Springer, 2011, iSBN 978-94-007-0256-1.

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Amsterdam, Aug 1974, pp. 471–475.

[3] "OpenDPI," Dec. 2011. [Online]. Available: http://opendpi.org/

[4] "Libpcap," Dec. 2011. [Online]. Available: http://www.tcpdump.org/

[5] Intel-Corporation, *Intel® 64 and IA-32 Architectures Developer's Manual: Combined Vols. 1, 2, and 3*, December 2011. [Online]. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-manual-325462.pdf

[6] E. Cannella, L. D. Gregorio, L. Fiorin, M. Lindwer, P. Meloni, O. Neugebauer, and A. D. Pimentel, "Towards an ESL Design Framework for Adaptive and Fault-tolerant MPSoCs: MADNESS or not?" in *Proceedings of the 9th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'11)*, Taipei, Taiwan, October 13-14 2011.

[7] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[8] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[9] ISO, "The ANSI C standard (C99)," ISO/IEC, Tech. Rep. WG14 N1124, 1999. [Online]. Available: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf

[10] S. P. Harbison and G. L. Steele, *C: A Reference Manual*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[11] M. Barr, *Programming Embedded Systems in C and C++*, 1st ed., A. Oram, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1998.

[12] "GNU Compiler Collection," Dec. 2011. [Online]. Available: http://gcc.gnu.org

[13] "ICD-C Compiler framework," Dec. 2011. [Online]. Available: http://www.icd.de/es/icd-c/icd-c.html