Efficient and Scalable OpenMP-based System-Level Design

Alessandro Cilardo, Luca Gallo, Antonino Mazzeo, and Nicola Mazzocca
University of Naples Federico II
Department of Electrical Engineering and Information Technologies
via Claudio 21, 80125, Naples, Italy
contact email: acilardo@unina.it

***Abstract*—In this work we present an experimental environment for electronic system-level design based on the OpenMP programming paradigm. Fully compliant with the OpenMP standard, the environment allows the generation of heterogeneous hardware/software systems exhibiting good scalability with respect to the number of threads and limited performance overheads. Based on well-established OpenMP benchmarks, the paper also presents some comparisons with high-performance software implementations as well as with previous proposals oriented to pure hardware translation. The results confirm that the proposed approach achieves improved results in terms of both efficiency and scalability.**

I. Introduction

This work addresses the adoption of OpenMP [1] for the high-level description of applications in electronic system-level (ESL) design. Ideally suited for medium granularity, loop-level parallelism, OpenMP has been extremely popular in the parallel computing community for years as a de-facto standard for parallel programming. Interestingly, the OpenMP programming model naturally meets the characteristics of current multi-processors systems-on-chip (MPSoCs), as it provides enough semantics to express parallelism explicitly, especially for data-intensive applications. During the recent years, a few works have addressed the use of OpenMP for programming MPSoCs [2], the generation of hardware components from OpenMP descriptions [3], [4], [5], or the integration of hardware accelerators in purely software OpenMP applications [6]. While exploring some facets of the adoption of OpenMP in non-standard computing platforms, however, none of these works propose a mature methodology and toolchain for the automated synthesis of heterogeneous hardware/software systems starting from an OpenMP application.

In this work, we present an experimental environment for OpenMP-based system-level design. The environment is fully compliant with the OpenMP standard, enabling the reuse of a large body of OpenMP code and kernels, including multicore applications, and it enables the integration with ESL design [7] and high-level synthesis techniques [8]. To demonstrate the effectiveness of our approach, we relied on the well-known EPCC benchmark suite [9]. Comparisons with high-performance pure software implementations show that the overheads incurred by our environment are promisingly lower and, importantly, they grow slower than the pure software solution as the number of threads is increased. We also compared our results with a previous work providing OpenMP-to-hardware translation [3], which was not focused on a system-level approach. Again, the comparisons show that the proposed approach achieves much better results in terms of scalability, a fundamental requirement in the parallel application domain.

The paper is organized as follows. Section II presents the background of this work. Section III describes the proposed approach and our prototypical environment. Section IV discusses the results of our work and presents some comparisons. Section V concludes the paper with some final remarks.

II. Background

The parallelism in OpenMP is based on the *fork-join* execution model, where a program is initialized as a single thread, the *master thread*, executed sequentially until the first parallel construct is encountered. This causes the master thread to create a team of threads executing the statements in the parallel section concurrently. An implicit barrier at the end of the parallel section makes the parallel threads synchronize with each other, after which only the master thread continues its execution. For a complete description of the constructs, clauses, and API functions mentioned in this paper, please refer to the OpenMP specification v3.1 [1]. Due to the popularity of OpenMP and its powerful model allowing an easy description of parallel high-performance applications, a few works have recently addressed the generation of hardware components from OpenMP programs, or the integration of hardware accelerators in purely software OpenMP applications. The work in [3] created backends that generate either synthesizable VHDL or high-level Handel-C code from OpenMP C programs, targeting an FPGA platform and presenting some quantitative results from their tool. The approach is basically oriented to pure hardware synthesis where each thread corresponds to a finite state machine. There is no explicit memory hierarchy, limiting the available memory to the resources available on chip and making it difficult to support the shared memory in a scalable and efficient way. Few details are provided on synchronization, while nested parallelism and dynamic scheduling, a part of the OpenMP specification, are not supported. Furthermore, only the integer data type is available for use in the OpenMP description. The authors of [4] presented the possibilities and limitations of the hardware synthesis of OpenMP directives. The work concludes that OpenMP is relatively "hardware friendly" due to the formalisms allowing an easy description of explicit parallelism. However, it excludes the full support

978-3-9815370-0-0/DATE13/©2013 EDAA

for the OpenMP standard, e.g. it drops dynamic scheduling of threads, an essential aspect of OpenMP, dynamic loop bounds in `for` loops, and typically software routines such as those related to time. While simplifying the implementation of approaches to OpenMP-based hardware-level design, the restrictions assumed by the above works limit the semantics available to parallel programmers and, even worse, they prevent the reuse of already available OpenMP programs and kernels.

Other works cannot be directly applied to the hardware synthesis process. For example, the work in [5] presented a methodology for transforming an OpenMP program into a functionally equivalent SystemC description. The emphasis is mainly on the source-to-source transformation process, as the output code is not necessarily guaranteed to be compliant with the OSCI SystemC Synthesizable Subset. The work in [2] introduced the support for a subset of OpenMP in a resource-constrained MPSoC platform. The paper is mainly focused on the challenges of implementing OpenMP on a multi-core system with no operating system, and does not cover the generation of custom hardware/software architectures. Finally, the authors in [6] present some extensions to OpenMP and a related runtime system implementation to specify the offloading of tasks to reconfigurable devices and the interoperability with OpenMP. Their contribution essentially relies on a host-based model where one or more FPGA boards act as accelerators providing functions to an OpenMP application. The approach assumes already existing reconfigurable binary code and mainly targets techniques to hide the details of the configuration process and movement of data. The design of the hardware accelerators does not involve high-level languages nor OpenMP in itself.

As emphasized by the above review, all the previous proposals are either focused on the integration of hardware accelerators in essentially software OpenMP programs, or on the pure hardware translation. In fact, the complexity and cost of a pure hardware implementation requires dropping the support for high-level features such as dynamic scheduling and nested parallelism. This choice tends to reduce OpenMP to a mere design-entry language for newly developed, small-scale hardware projects, inhibiting the support for real-world OpenMP applications, possibly developed with no hardware requirements in mind. A few additional limitations in hardware-oriented OpenMP solutions include the use of centralized mechanisms for controlling interactions among threads, causing scalability issues as the number of threads is increased, limited support for external memory infrastructures and efficiency-critical mechanisms such as caching, and unsupported runtime library routines whose parameters cannot be evaluated statically at compile time.

III. An environment for OpenMP-based electronic system-level design

Most limitations of the previous hardware-oriented OpenMP solutions [3], [4] derive from the fact that they target the generation of purely hardware components, identifying a subset of OpenMP as a design-entry language in a high-level synthesis flow. We aim at overcoming the above restrictions and envision a scenario where OpenMP is fully supported and adopted as a mature formalism for *system-level* design. The essential aspect of the proposed methodology is the generation of heterogeneous systems, including one or more processors and dedicated hardware components, where OpenMP threads can be mapped to either software threads or hardware blocks. The control-intensive facets making the full OpenMP difficult to implement in hardware are managed in software, while the data-intensive (and performance critical) part of the OpenMP application is addressed by dedicated software/hardware parallelism. Hardware threads are generated by means of standard high-level synthesis tools that perfectly fit the structure of an OpenMP program, where the application logic is still described by means of plain C code. On one hand, this approach provides full support for standard-compliant OpenMP applications, as well as fundamental "general-purpose" characteristics such as memory hierarchies and management. On the other hand, it naturally meets the emerging trends for heterogeneous hardware/software MPSoC and embedded systems, where the presence of one or more instruction processor cores is now mainstream.

The proposed environment targets the integration with standard flows for system-level design of heterogeneous systems. Such flows normally include a hardware/software partitioning stage [10], either performed manually or supported by automated tools, resulting in the definition of the physical architecture and providing inputs to two concurrent branches covering the software compilation and the high-level hardware synthesis. These re-join together on the system composition step, where the whole system is built, usually assembling hardware/software library components and application-specific components generated by the previous steps, followed by the low-level hardware synthesis. The proposed methodology is used to automate the stages following the hardware/software partitioning choices, as it covers the transformation of the C/OpenMP code as needed by the subsequent high-level synthesis step, and the extension of the application code – both hardware and software– with structures enabling the interaction of the different components.

A. Reference system architecture

The parallel heterogeneous architecture is defined in such a way as to orchestrate its hardware/software components in a distributed fashion, avoiding centralized hardware elements. Figure 1 exemplifies the main components of the architectural model adopted by the proposed environment. Each subsystem represents an OpenMP thread, executing a certain portion of the original OpenMP code. Software subsystems are processors for which a portable C code (derived from the OpenMP application) can be compiled. Hardware subsystems are generated with HLS in order to execute parts of the original parallel code, or they can be ordinary peripherals such as timers, non-volatile memory blocks containing boot code, or ad-hoc arithmetic components [11]. The OpenMP *main thread* is mapped to a software subsystem, while the other threads can be mapped to either hardware or software subsystems. Currently, we only support one thread per subsystem, but as a natural extension we envision a scenario where several OpenMP threads share the same processor using a multithreaded OS kernel. The figure also depicts the memory infrastructure. The memory components may be implemented in a different technology and indeed they may be mapped to FPGA internal RAM blocks or off-chip SDRAM memory, enabling the synthesis of real-world systems

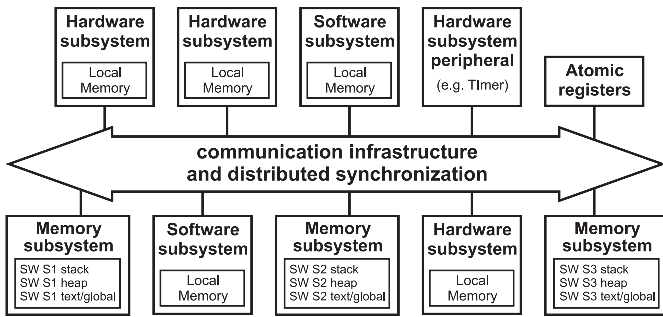

Fig. 1. Architecture of a heterogeneous hardware/software system generated from an OpenMP program

working on large amounts of data. Shared memory areas are accessed by all subsystems (currently, the environment does not provide memory protection mechanisms). Each hardware subsystem has its own local memory corresponding to the synthesized registers, while software subsystems can also have a local memory corresponding to the processor cache levels of the memory hierarchy. The atomic registers, accessible on a particular address by all subsystems, allow the implementation of *read-only-after-write* primitives: if a read is performed without a previous write, an error value is returned. This device acts as the basic building block for implementing synchronization directives. The number of instantiated atomic registers is equal to the number of `barrier`, `atomic`, and `critical` constructs plus the numbers of calls to `omp_init_locks()` (that creates a lock) found in the original OpenMP code. Special emphasis was put on the support for the `shared` and `private` clauses, which are essential in OpenMP because of its shared memory model. Shared variables are detected by tracking each access and replacing it with suitable memory operations accessing the appropriate local/remote areas.

Hardware subsystems generated by HLS are memory mapped, have each their own thread id (`tid`), have DMA master access capabilities, and include a `Start`/`Done` synchronization port. A natural approach for connecting `Start`/`Done` signals would rely on a centralized scheme, where the master thread sends and receives all signals. This scheme preserves design simplicity but may easily compromise scalability. To overcome this problem, we exploited a fully distributed approach where the OpenMP threads involved in a fork-join structure form a tree. Each node in the tree, i.e. a thread, forwards a `Start` signal to its subtrees before starting its own computation, and forwards a `Done` signal to its parent node only after receiving all the subtrees' `Done` signals and completing its own task. As a consequence, the implicit barrier at the end of OpenMP work-sharing constructs takes a time corresponding to the worst-case propagation delay through the tree, which is logarithmic in the number of threads. The same mechanism described above is used to implement explicit `barrier` constructs.

B. OpenMP constructs and API functions

Due to the lack of space, we will not provide here the full technical details of the implementation of each OpenMP clause and API function. We will only give a couple of examples. The first one is the implementation of the `firstprivate` clause that deals with variable scoping and data initialization. Below is the algorithm used for the master thread and the other threads to support the `#pragma omp parallel firstprivate (var)` construct:

Master thread:

```
- Copy the values of variables listed as firstprivate
    to the shared memory to a predefined offset
- Send the Start signal to the child threads
- Parallel section
- Wait for the Done signal from child threads
```

Other threads:

```
- Wait for the Start signal from the parent thread
- Send Start signal to the children threads
- Take values from shared memory and initialize private vars
- Parallel section
- Wait for the Done signals from the children threads
- Send the Done signal to the parent thread
```

The additional overhead caused by the `firstprivate` clause only depends on the number of listed variables and does not increase with the number of threads, preserving the application scalability. Another example is the support for dynamic scheduling. Following is the code each thread needs to execute to dynamically parallelize a `for` cycle.

```
index=0;
while (index  < total_iterations_number){
  temp = iter;                      //  lock on 'iter'
  if (temp != index ){
      index = temp;
  }
  if (temp != error_code && temp < total_iterations_number){
      prev_iter = temp;
      iter = prev_iter+chunkSize;  //  unlock on 'iter'
      for (i=prev_iter; i<=prev_iter+chunkSize; prev_iter++){
         // original code of for iterations
      }
      index = temp+chunkSize;
  }
}
```

where `iter` is an atomic memory location (see Section III-A) and `prev_iter`, `temp`, and `index` are local variables. As can be seen, the support is completely distributed. Consequently, the threads execute a number of iterations determined at run time according to the computational power of the unit they are allocated to and the different load they happen to handle, fully implementing the semantics of the `dynamic` clause. A further example is the support for the `omp_get_thread_num()` function. To support its implementation, the thread identifier `tid` is passed to the subsystem along with the `Start` signal, so that each call to the above function can be directly replaced with the value of `tid`.

C. Implementation details

The main element of our prototypical environment is an ad-hoc lightweight compiler dealing with source-to-source transformation. The compiler, built on top of a standard C grammar with OpenMP extensions, was implemented in C/C++ and relied on the well-known *Flex* and *Bison* tools [12] for the generation of the lexical scanner and the parser, respectively. The compiler takes C source code with OpenMP `#pragma` statements as input, generating source files suitable for high-level synthesis and for the compilation on the target embedded microprocessors. Furthermore, it extends the code with suitable software structures used to manage the communication and the synchronization between subsystems, based on the techniques introduced in the previous subsections. Concerning the HLS, we relied on Impulse CoDeveloper [13], while for the

platform-based system composition we adopted the Embedded Development Kit (EDK) [14]. Currently, our prototypical environment only supports Xilinx FPGA devices and Microblaze processors for the implementation of the software subsystems.

IV. Results

This section provides some quantitative results to assess the effectiveness of the proposed approach. The evaluation of experimental results was focused on measuring the overhead related to the implementation of the OpenMP constructs rather than the absolute execution times, which are mainly depending on the underlying technology. Precisely, we measured the overhead/execution time ratio, i.e. the *normalized overhead*. To this aim, we relied on the well-known EPCC benchmarks [9], designed to provide performance comparisons between OpenMP implementations and measure the overheads of OpenMP constructs, including parallel regions, synchronization, loop scheduling, and data scoping. We compared the normalized overhead with an OpenMP implementation for a Windows 7 OS on an Intel i7 processor at 1.8 GHz running the same benchmarks. The first table below summarizes the overhead trends for some constructs, giving the normalized overhead average value over 2 to 10 threads along with its average slope, i.e. the additional overhead per thread. The important clue is that, for every construct we consider, in addition to being low in its absolute values, the overhead grows very slowly, confirming both the efficiency and the scalability of the proposed approach. Some constructs like `#pragma omp single` and `#pragma omp master` are supported with a zero overhead because they are eliminated in the code transformation phase as they do not correspond to any run-time mechanism. To further emphasize the impact of our approach, we also compared our results with [3], which in fact is the only solution for OpenMP-to-hardware translation presenting a working tool and some performance figures, used here for comparisons. These include the speed-up for the Sieve of Eratosthenes algorithm (comparisons are given in the second table below) and the clock frequency for an Infinite Impulse Response (IIR) filter (comparisons are given in the third table below in terms of the average frequency and the average *frequency loss* per thread in MHz). The tables confirm that our approach achieves considerably improved levels of scalability, impacting both the application speed-up and the complexity of the generated system, which sustains higher frequencies as the number of threads is increased.

EPCC: Normalized Overhead (Average value / Slope), 2 to 10 threads			
Construct	**FPGA**	**Intel i7**	**Gain**
`Private`	9.7% 0.6%	61.6% 12.1%	6.35 20.2
`Firstprivate`	70.1% 0.6%	84.1% 14.7%	1.2 24.5
`Dynamic`	28.3% 3.3%	89.5% 8.2%	3.16 2.5
`Static`	6.9% 1.6%	72.6% 13.0%	10.5 8.1
`Critical`	2.8% 0.8%	7.4% 1.8%	2.64 2.25

Sieve of Eratosthenes: Speed-up, 2 to 8 threads			
Threads	**here**	**[3]**	**Gain**
2	1.97	1.32	1.49
4	3.85	2.19	1.76
6	5.40	2.03	2.66
8	6.64	2.38	2.79

IIR: Average Frequency / Frequency loss per thread, 2 to 8 threads			
	here [MHz]	**[3] [MHz]**	**Gain**
	166 1.29	56 8.6	2.96 6.67

V. Conclusions

The paper described an experimental approach to electronic system-level design based on the OpenMP programming paradigm. The ad-hoc compiler we developed covers the full OpenMP specification and introduces several new techniques at the architectural level matching the requirements of the parallel OpenMP constructs. The work presented some details of our prototypical environment and provided efficiency and scalability results obtained from well-established benchmarks. The results proved that OpenMP may represent a promising path to system-level design for the emerging class of heterogeneous system-on-chip architectures.

Acknowledgment

The authors would like to thank Impulse Accelerated Technologies for providing access to the Impulse CoDeveloper high-level synthesis tool.

References

[1] OpenMP Architecture Review Board. (2011) OpenMP application program interface, v3.1. [Online]. Available: www.openmp.org

[2] W.-C. Jeun and S. Ha, "Effective OpenMP implementation and translation for multiprocessor System-on-Chip without using OS," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference - ASP-DAC '07*, Jan. 2007, pp. 44–49.

[3] Y. Leow, C. Ng, and W. Wong, "Generating hardware from OpenMP programs," in *IEEE International Conference on Field Programmable Technology (FPT 2006)*, Dec. 2006, pp. 73–80.

[4] P. Dziurzanski and V. Beletskyy, "Defining synthesizable OpenMP directives and clauses," in *Proceedings of the 4th International Conference on Computational Science - ICCS 2004*, ser. LNCS, vol. 3038. Springer, 2004, pp. 398–407.

[5] P. Dziurzanski, W. Bielecki, K. Trifunovic, and M. Kleszczonek, "A system for transforming an ANSI C code with OpenMP directives into a SystemC description," in *Design and Diagnostics of Electronic Circuits and Systems, 2006*. IEEE, apr 2006, pp. 151–152.

[6] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jimenez-Gonzalez, "OpenMP extensions for FPGA accelerators," in *International Symposium on Systems, Architectures, Modeling, and Simulation, 2009 - SAMOS '09*, Jul. 2009, pp. 17–24.

[7] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification. A Prescription for Electronic System-Level Methodology*. Elsevier Inc, 2007.

[8] P. Coussy and A. M. (Eds.), *High-Level Synthesis from Algorithm to Digital Circuit*. Springer, 2008.

[9] EPCC. (2012) EPCC OpenMP benchmarks. [Online]. Available: http://www.epcc.ed.ac.uk/software-products/epcc-openmp-benchmarks/

[10] A. Cilardo, P. Durante, C. Lofiego, and A. Mazzeo, "Early prediction of hardware complexity in HLL-to-HDL translation," in *International Conference on Field Programmable Logic and Applications (FPL 2010)*, Aug. 2010, pp. 483–488.

[11] A. Cilardo, "A new speculative addition architecture suitable for two's complement operations," in *Design, Automation and Test in Europe Conference DATE'09*, Apr. 2009, pp. 664–669.

[12] V. Das, *Compiler Design Using FLEX and YACC*. Prentice-Hall of India Pvt.Ltd, 2007.

[13] Impulse Accelerated Technologies. (2012) Impulse CoDeveloper. [Online]. Available: http://www.impulseaccelerated.com

[14] Xilinx. (2012) Platform studio and the embedded development kit (EDK). [Online]. Available: http://www.xilinx.com/tools/platform.htm