Performance Modeling of Parallel Applications on MPSoCs

Marco Lattuada, Christian Pilato, Antonino Tumeo, Fabrizio Ferrandi
Dipartimento di Elettronica e Informazione, Politecnico di Milano
Via Ponzio 34/5, 20133 Milano, Italy
{lattuada,pilato,tumeo,ferrandi}@elet.polimi.it

***Abstract*—In this paper we present a new technique for automatically measuring the performance of tasks, functions or arbitrary parts of a program on a multiprocessor embedded system. The technique instruments the tasks described by OpenMP, used to represent the task parallelism, while ad hoc pragmas in the source indicate other pieces of code to profile. The annotations and the instrumentation are completely target-independent, so the same code can be measured on different target architectures, on simulators or on prototypes.**

We validate the approach on a single and on a dual LEON 3 platform synthesized on FPGA, demonstrating a low instrumentation overhead. We show how the information obtained with this technique can be easily exploited in a Hardware/Software design space exploration tool, by estimating, with good accuracy, the speed-up of a parallel applications given the profiling on the single processor prototype.

I. Introduction

Performance analysis is a crucial phase of the embedded design flow. Multi-Processor Systems-on-Chip (MPSoCs), composed of several processors, often heterogeneous, interconnected with memories, application specific accelerators and Input/Output components, are the standard architectures for modern embedded systems [1].

The complex organization of these architectures, the parallelism, the synchronization and the communication mechanisms introduce several issues in the optimization process. In particular, accurate measurements or estimations of the execution time of the applications are required to help the designer in meeting the performance constraints. However estimation methodologies cannot be used if some parts of the application are provided only within libraries, or the input data are available only on the target architecture. In such cases, the only way to gather accurate performance information is to directly instrument the code onto the target architecture.

In this paper, we propose a technique for automatic code instrumentation of parallel applications for MPSoCs. The technique is selectively applied only to the most interesting parts of the code, identified by pragma annotations. The parallelism is described through OpenMP [2], while ad hoc annotations are used to measure the execution times of functions and arbitrary parts of code. Several methodologies and tools have been proposed for code instrumentation and analysis.

In [3] the authors apply a fine-grained instrumentation to estimate the performance of embedded applications inside a Transaction Level Model (TLM) tool for MPSoC design. The instrumentation sums the clock cycles required by a specific instruction each time it is executed. The execution delay of each instruction, in clock cycles, has to be provided to the tool. Garcia *et al.* present Pet [4], a tool for monitoring and analyzing parallel applications on embedded multiprocessors. The tool analyzes the execution trace and identifies all the interactions among the tasks and their occurrence. However, it only works on a specific embedded architecture and cannot manage general parallelism annotations like OpenMP. Some works discuss the performance analysis of OpenMP parallel application in the High Performance Computing field. SCALEA [5] allows the evaluation of the overheads introduced by parallel programming paradigms. It deals not only with OpenMP but also with Message Passing Interface (MPI) and High Performance Fortan (HPF). Nevertheless, SCALEA is tightly integrated with its Fortran compiler framework, and is geared towards distributed systems. OPARI [6] is a source-to-source translation tool that exploits the idea of OpenMP pragma/directive rewriting, automatically adding all the necessary calls to a runtime measurement library. The tool, however, do not allow to measure specific parts of the code. We follow the ideas proposed in those solutions for the performance analysis of parallel OpenMP code, but introduce those techniques on embedded architectures. Konkin *et al.* in [7] describe a methodology to identify the points where to add instrumentation for performance measurement. The authors define as suitable instrumentation points the interfaces among different software modules, like function call points. This solution allows complete automatic instrumentation of the source code, but the large number of instrumentation points introduces significant overheads. Our proposal, instead, allows selecting which points to instrument. Our solution is somewhat inspired by DTrace [8]. With Dtrace, the code is instrumented by the programmer during the development, and the profiled data are collected at runtime through the use of scripts that enables the probes. In our case, the developer specifies with pragmas what he or she wants to measure, and at compilation time the tool inserts the required instrumentation. Our approach is thus more suitable for embedded systems: performance optimization is usually done at design time, and not adding performance libraries and probes (even if only activated when required) reduces the memory footprint of the application.

Moreover, embedded system design imposes further constraints. Often, during the designs of the applications, we can rely the code on the final target platform, but only on a prototype which does not represent all the details of the final architecture. Usually, for example, we have FPGA prototypes that, for area reasons, have few processors than the final ASIC architecture. Thus, solutions able to correctly estimate the speed up and the overhead due to the higher degree of parallelism are required. Our methodology can be used for

the estimation of the performance of parallel code, starting from a single processor solution.

The contributions of this paper can be summarized as follows:

- it introduces a technique for measuring OpenMP partitioned applications on embedded systems;
- it proposes a fast, target independent technique for arbitrary code instrumentation, that, limiting the overheads only on the interesting parts of the code, may be adopted in a embedded design flow;
- it uses the instrumentation technique to estimate the performance of a dual processor embedded platform, starting from a single processor solution.

The paper is organized as follows. Section II describes the proposed technique. Section III describes our case study, while Section IV discusses the experimental results. Finally, Section V concludes the paper.

II. Proposed methodology

The proposed methodology is integrated in Zebu, a tool provided in PandA [9], a framework for hardware/software co-design. The methodology consists in a performance analysis flow composed of three different phases, that will be detailed in the following:

- *instrumentation* (Section II-1): the original source code is parsed, analyzed and then reproduced with the instrumentation added in the proper points;
- *compilation* (Section II-2): the produced source code is compiled and executed on the target architecture;
- *data collection* (Section II-3): the measured data are collected and associated with the parts of the code which they refer to.

1) Instrumentation: Zebu exploits the GNU/GCC compiler to translate the initial C source code into the GIMPLE intermediate representation, considering three different type of pragmas:

- *OpenMP pragmas*: we consider only the *omp parallel sections*, *omp sections* and *omp parallel for* pragmas;
- *function pragmas*: they identify the functions we are interested to measure the execution time;
- *custom measurement pragmas*: they identify arbitrary blocks of structured code that may require measurement.

The intermediate representation is then modified adding the requested instrumentation. A unique identifier is assigned to each *omp parallel sections* or *omp sections* pragma, when encountered, and the instrumentation code is added immediately before and after the corresponding code block. Next, a unique identifier is assigned also to each *omp section* in the *sections* and the instrumentation is added at its beginning and at its end. In this way, it is possible to measure both the execution time of each task of the *omp sections* region and the overall synchronization costs. The *omp for* pragmas are treated in two different ways. When we have the number of threads active at that point , the pragma is replaced with an *omp parallel sections* region and loop iterations are statically partitioned and assigned to different *omp section*. When the number of threads is unknown, the *omp for* pragma is treated like a *custom measurement pragma* and only the execution time of the whole loop is measured.

The second type of considered pragmas is the *function pragma* which is associated with the declaration of the functions. When a call point of an annotated function is detected, a unique identifier is associated to that and the instrumentation code is added immediately before and after the call. Instrumenting the code in this way allows distinguishing the execution time of the function according to the call points. Furthermore, the overhead of the function call is also considered in the execution time.

Finally, when a *custom measurement pragma* is found, the instrumentation code is simply added at each entry and exit point of the annotated code block. Also in this situation, we associate a unique identifier with the portion of code, adding the related instrumentation code to the produced code.

About the instrumentation, Zebu adds a numeric array, which stores the measures, and some function calls to record the application overall execution time and to allow collecting the data at the end of the execution. When all the needed instrumentation code has been added, the GIMPLE code is translated back to C source code.

2) Compilation: Since the instrumentation code added in the previous phase is completely target-independent, in this phase, it is customized for a particular target through an architecture-specific definition file composed of two different parts. The first part defines the type of the array elements which record the measures. The second part contains the implementation of the functions which effectively measure the execution time, since these implementations heavily depend upon both the considered architecture and the operating system, if present. The instrumented source code and the architecture-specific definition file are compiled and then linked, obtaining the executable object code for the application on the specific target architecture.

3) Data Collection: The last phase of the flow consists in collecting the measurement data by executing the application onto the target architecture. Different runs with different inputs have to be executed if the application is strongly data-dependent, producing different datasets to be collected. Note that, if a piece of code is executed more than once during a single run of the application, the technique measures the related average execution time. If the application has been executed more than once, the average execution time of each annotated code section is computed. Finally, since Zebu maintains the correspondence between the unique identifiers and the parts of the code, it easily assigns to each task, annotated function or annotated part of code, its measured execution time.

III. Case study: LEON 3 MP

In this section, we show how the proposed technique is applied to LEON 3 based systems [10]. In particular, we demonstrate that our approach allows the performance estimation of a parallel application, annotated with OpenMP, on a multiprocessor system, while measuring its execution time on a single processor architecture.

We implemented two different architectures. The first uses a single LEON 3, the second integrates two processors. In both the designs, we enabled the Memory Management Unit (MMU) and instantiated 16 KB of instruction cache and 8 KB of data cache. We enabled the Gaisler Floating Point Unit

```
void loop(int nthreads, int size, int numiter) {
  /* Variable Declaration and initialization code*/
  ...
  for(iter=0; iter<numiter; iter++) {
    #pragma omp parallel for ...
    shared(V,oldV,totalSize) schedule(static)
    for (i=0; i<totalSize; i++)
       oldV[i] = V[i];
    #pragma omp parallel for ...
    for (i=0; i<totalSize-1; i++)
       V[i] = f(V[i],oldV[i+1]);
  }
}
```

Fig. 1. The `loop` function of *Loop with dependencies* benchmark

(FPU) Lite on both the systems, instantiating a FPU per core on the multiprocessor configuration. Floating point operations are required by the standard implementation of *libgomp* (the OpenMP library). On multiprocessor LEON 3 solutions, the data caches are coherent. We implemented the architectures on a Xilinx Virtex-5 FX70T FPGA, with a target frequency of 80 MHz, using Xilinx ISE 10.1 for synthesis, mapping and routing. With the MMU support, LEON 3 can run Linux, in single and multi-processing mode. Both the system used the Snapgear distribution, with the Linux kernel version 2.6.21.1.

We now describe the steps required to perform the speed-up estimation on an example, the first version of the *Loop with Dependencies* benchmark from OpenMP Source Code Repository [11]. The significant part of its source code is reported in Figure 1.

The function *loop*, which is the kernel of the application, has two *omp parallel for* regions. The number of threads executing the application is two (the number of processor of our target architecture), so each *omp parallel for* region is replaced by an *omp parallel sections* with two *section*s. The resulting instrumented source code is shown in Figure 2. The `START_PARALLEL` and `STOP_PARALLEL` functions measure the execution times of the two parallel regions identified as 1 and 4. The `START_SECTION` and `STOP_SECTION` functions, instead, measure the execution time of the four *section*s identified as 2, 3, 5 and 6.

The instrumented application is then compiled with the architecture-specific definition file for sequential execution, ignoring the OpenMP pragmas, and is executed on the single processor platform. In particular, the functions implemented

```
void loop(int nthreads, int size, int numiter) {
  /* Variable Declaration and initialization code */
  ...
  for(iter=0; iter<numiter; iter++) {START_PARALLEL(1);
    #pragma omp parallel sections num_threads(2) ... {
      #pragma omp section {START_SECTION(2);
        for (i=0; i<totalSize/2; i++)
          oldV[i] = V[i];
        STOP_SECTION(2);}
      #pragma omp section {START_SECTION(3);
        for (i=totalSize/2; i<totalSize; i++)
          oldV[i] = V[i];
        STOP_SECTION(3);}
    } STOP_PARALLEL(1);
    START_PARALLEL(4);
    #pragma omp parallel sections num_threads(2) ... {
      #pragma omp section {
        START_SECTION(5);
        for (i=0; i<(totalSize-1)/2; i++)
          V[i] = f(V[i],oldV[i+1]);
        STOP_SECTION(5);}
      #pragma omp section {START_SECTION(6);
        for (i=(totalSize-1)/2; i<totalSize-1; i++)
          V[i] = f(V[i],oldV[i+1]);
        STOP_SECTION(6);}
    } STOP_PARALLEL(4);
  }
}
```

Fig. 2. The instrumented source code of `loop` function of *Loop with dependencies* benchmark

into the architecture-specific definition file for LEON 3 based systems are based on the Linux system function *gettimeofday*. At the end of the application execution, Zebu annotates each task of the application with the measured execution time. The next step consists in effective estimation of parallel execution time, accomplished the differences among the sequential and the parallel execution. On the parallel architecture, the execution time cp_P of each parallel region P composed by task $t \in P$ can be computed as:

$$cp_P = c_f + max_{t \in P} c_t + c_j \quad (1)$$

where c_f is the *fork* cost, c_t the execution time of task t, and c_j is the *join* cost. On the other hand, in the sequential execution, the time cs_P needed to execute the code of the same parallel region P is $cs_P = \sum_{t \in P} c_t$. So the time saved by executing the parallel region onto a multiprocessor architecture (g_P) can be estimated as:

$$g_P = cs_P - cp_P = \sum_{t \in P} c_t - max_{t \in P} c_t - c_f - c_j \quad (2)$$

.

The execution times have to be combined with the profiling information to produce a correct estimation. Consider the *Loop with dependencies* example presented in Figure 2. The two parallel regions are the body of a loop which is executed $numiter$ times, so the g_P has to be multiplied by $numiter$.

Two aspects of a multi-processor system are not taken into account in estimating the execution time of the parallel application using sequential information: the contention in accessing shared resources, such as the memory, and the cache conflicts.

IV. Experimental Results

We validate the proposed technique on a set of benchmarks extracted from the OpenMP Source Code Repository[11] and from MiBench [12] on both the architectures described above. The OpenMP Source Code Repository[11] benchmarks are already annotated with OpenMP pragmas. The MiBench benchmarks have been parallelized by hand, splitting the kernels of each application into one or more pairs of parallel tasks through OpenMP annotations. In particular, the data obtained by the single-processor architecture have been used to estimate the execution on the multi-processor architecture. The number of instrumentation points for each benchmark is reported in the Table I. Each benchmark has been compiled with a GNU/GCC Sparc cross-compiler without optimization (*-O0*) and with the *-O2* optimization level.

Benchmark	Dataset	IP	MP
fft_6	(test)	20	308
jacob	(test)	8	8
Loops with Dependences	(test)	13	102
lu	(test)	14	14
mandel	(test)	8	8
MolecularDynamic	(test)	14	14
pi	(test)	8	8
basicmath	large	14	14
blowfish	(test)	14	14
dijkstra	large	8	8
jpeg encoder	input_small	14	258
edge detection	large	8	8
corner detection	large	8	8

TABLE I
CHARACTERISTICS OF THE BENCHMARKS. *IP* IS THE NUMBER OF INSTRUMENTATION POINT IN THE SOURCE CODE, *MP* IS THE NUMBER OF MEASURES PERFORMED DURING THE EXECUTION.

The results of all the executions are reported in the left part of the Table II. In the three columns labeled with *Sequential* we show the overhead introduced by the instrumentation for measuring the task performance on the first architecture (single-processor platform). In particular, in *Real* we report the execution time of the benchmark measured without instrumentation. *(*OH), instead, reports the overhead introduced by the instrumentation, that is usually very reduced (it ranges from 0.0% to 0.3%). In three cases (fft_6, *Loops with Dependencies* and *jpeg encoder*) the overhead is more relevant, due to a higher number of measures performed during the execution. In particular, in *Loops with Dependencies*, there are a huge number of measures with respect to the small size of the benchmark.

The central part of Table II shows the instrumentation overhead on the second architecture (dual-processor platform). *Real* and *OH* report the execution time without task performance instrumentation and the overhead introduced by the instrumentation, respectively. The instrumentation overhead is for most of the benchmarks bigger than what observed on the single-processor architecture, for mainly two reasons. First, the overall execution time of the applications on the dual processor platform is smaller than on the single LEON solution, so the impact of the instrumentation is more relevant. Second, the instrumentation in parallel tasks generates a contention while accessing the common structures used to perform the profiling.

It is worth noting the results of the *Loops with Dependencies* benchmark. In this case, the overhead due to the task creation/destruction/synchronization is bigger than the benefits introduced by the parallelization. Thus, the parallel applications results longer than the sequential one and, as a consequence, the relative instrumentation overhead of the parallel version results smaller.

The last two columns of the table show the results of the estimation of parallel execution computed using the method described in Section III. *Estim.* is the overall execution time estimated and *Error* is the error of the estimation. The cost for forks and joins has been obtained by applying our technique to the OpenMP MicroBenchmark Suite [13] and the error obtained in estimation ranges from 0.0% to 6.9%. Specifically, we observe that the parallel execution time is underestimated for all the benchmarks, depending on the contention for accessing shared resources and from the cache conflicts, as previously detailed in Section III.

		Execution Overhead				Est.
		Sequential		Parallel		Acc.
Benchmark	OL	Real(s)	OH(%)	Real(s)	OH(%)	Err.(%)
fft_6	O0	28.644	0.8	17.038	0.6	5.3
	O2	16.715	0.4	10.797	0.2	5.9
jacob	O0	98.305	0.0	73.832	0.0	0.7
	O2	42.304	0.1	35.229	0.5	2.4
Loops with	O0	0.230	2.6	0.591	2.1	1.1
Dependencies	O2	0.135	3.7	0.506	0.7	0.4
lu	O0	57.047	0.0	35.308	0.1	0.8
	O2	32.820	0.1	18.728	0.1	1.2
mandel	O0	69.983	0.0	35.214	0.0	1.0
	O2	38.725	0.0	19.386	0.0	1.8
Molecular	O0	82.272	0.0	44.356	0.0	0.0
Dynamic	O2	43.511	0.0	24.549	2.2	1.6
pi	O0	1.964	0.0	1.130	0.0	0.4
	O2	1.252	0.0	0.702	0.0	3.4
basicmath	O0	3.662	0.1	2.640	0.2	1.3
	O2	2.291	0.1	1.340	0.3	1.1
blowfish	O0	0.448	0.2	0.273	0.3	3.2
	O2	0.190	0.0	0.114	0.8	6.9
dijkstra	O0	14.206	0.0	9.314	0.1	2.5
	O2	8.405	0.3	5.995	0.9	1.7
jpeg encoder	O0	3.254	0.7	2.307	0.8	2.4
	O2	2.603	0.7	1.928	0.7	2.0
edge detection	O0	4.236	0.0	3.308	0.2	1.8
	O2	1.703	0.1	1.217	0.3	1.5
corner detection	O0	1.563	0.0	0.943	0.1	4.4
	O2	0.648	0.3	0.420	0.5	4.7
Maximum			3.7		2.2	6.9
Mean			0.4		0.5	2.3
Std. Deviation			0.9		0.6	1.8

TABLE II
RESULTS OBTAINED BY APPLYING THE PROPOSED TECHNIQUE.

V. CONCLUSION

In this paper we presented a technique to automatically measure the performance of the different parts of an application on a MPSoC. The technique uses different pragmas to identify the parts of the code to be measured and the related target-independent instrumentation is directly inserted into the source code. The proposed technique has been validated on a set of benchmarks for parallel and embedded computing on a LEON 3 platform. The results show that the overhead introduced is small and that the performance profiled on a sequential version of an application can be used to estimate the execution time of its parallel version.

Future works will focus on the extension of the proposed methodology to heterogeneous architectures, considering in particular the performance analysis of functions offloaded to hardware accelerators, and on refinements of the methodology for allowing the estimation of the effects of the resources contention.

ACKNOWLEDGMENTS

This research was partially funded by the European Community's Sixth Framework Programme, hArtes project (www.hartes.org).

REFERENCES

[1] W. Wolf, "The future of multiprocessor systems-on-chips," in *DAC*, 2004, pp. 681–685.
[2] OpenMP, "Application Program Interface, version 2.5," May 2005. [Online]. Available: http://www.openmp.org
[3] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A sw performance estimation framework for early system-level-design using fine-grained instrumentation," in *DATE '06*, 2006, pp. 468–473.
[4] J. Garcia, J. Entrialgo, D. F. Garcia, J. L. Diaz, and F. J. Suarez, "Pet, a software monitoring toolkit for performance analysis of parallel embedded applications," *J. Syst. Archit.*, vol. 48, no. 6-7, pp. 221–235, 2003.
[5] H. L. Truong, T. Fahringer, G. Madsen, A. D. Malony, H. Moritsch, and S. Shende, "On using scalea for performance analysis of distributed and parallel programs," in *Supercomputing '01*. New York, NY, USA: ACM, 2001, pp. 34–34.
[6] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and prototype of a performance tool interface for openmp," *J. Supercomput.*, vol. 23, no. 1, pp. 105–128, 2002.
[7] D. P. Konkin, G. M. Oster, and R. B. Bunt, "Exploiting software interfaces for performance measurement," in *WOSP '98*. New York, NY, USA: ACM, 1998, pp. 208–218.
[8] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX Annual Technical Conference*, 2004.
[9] "The PandA framework." [Online]. Available: http://trac.elet.polimi.it/panda
[10] "Leon 3 Processor." [Online]. Available: http://www.gaisler.com
[11] A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano, "The openmp source code repository," in *PDP*, 2005, pp. 244–250.
[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC*, 2001, pp. 3–14.
[13] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," in *In Proceedings of First European Workshop on OpenMP*, 1999, pp. 99–105.