Abstract— There is some consensus that Embedded and HPC domains have to create synergies to face the challenges to create, maintain and optimize software for the future many-core platforms. In this work we show how some HPC performance analysis methods can be successfully adapted to the embedded domain. We propose to use Virtual Prototypes based on Instruction Set Simulators to produce trace files by transparent instrumentation that can be used for post-mortem performance analysis. Transparent instrumentation on ISS kills two birds in one shot: it adds no overhead for trace generation and it solves the problem of trace storage.

A virtual prototype is build to generate OTF traces that are later analyzed with Vampir. We show how the performance analysis of the virtual prototype is valuable to optimize a parallel embedded test application, allowing an acceptable speedup factor on 4 processors to be obtained.

Keywords - MPSoC; trace generation; NIOS; FPGA; NoC; Virtual Prototypes; Performance Analysis

I. INTRODUCTION

According to the ITRS roadmap [1] the increasing capacity of integration is not being accompanied by an increase in the capacity to design and verify. This creates a productivity gap, which was first dominated by the design phase. But since larger blocks (like processors and memories) are reused to create complex MPSoCs, the design effort will become secondary in front of the verification effort, which will receive the main focus. And again this verification will be probably not dominated by hardware verification (either at RTL or TLM level) but by parallel or concurrent software. With the imminent widespread of many-core processors, the verification of embedded systems will become a harder issue. HPC community has already faced this problem for decades, so we should try to identify the points of connection between both worlds and adapt the tools to be incorporated in the development process of future embedded systems.

In this paper we present a new way to combine some of the already used methods in the Embedded and HPC world, to optimize parallel applications for Embedded MPSoCs. This approach allows analyzing MPSoC software performance with a high time accuracy even if the hardware system is not fully available.

The remainder of this paper is organized as follows. In the following section we will review some of the common performance analysis techniques used by the HPC community. In section III we elaborate the idea of performance analysis by transparently instrument an Instruction Set Simulator, which is explained in more detail in section IV. In section V we give some results obtained by analyzing selected applications, and we finally end with conclusions.

II. HPC PERFORMANCE ANALYSIS

Performance analysis is the basis of the performance optimization process for the HPC community. The parallelization process usually starts with a serial application that is analyzed, either by hand or semi-automatically, to detect the regions of the code that contain potential parallelism. The way to generate code that exploits the potential parallelism is out of the scope of this work, but there are some simple methods to avoid excessive efforts. A fairly easy way of parallelizing is the usage of OpenMP compiler directives. That way the compiler takes care of managing the threads needed for the code to work in parallel. MPI (Message Passing Interface) is another method, that in contrary to OpenMP, does not rely on shared memory to work but requires more programming effort as it is a communication protocol where the messages have to be programmed explicitly by the developer.

In both popular models, tools exist to address the need of identifying computational and communication bottlenecks. A usual approach is to instrument the application to produce traces that can be later analyzed to find those bottlenecks. Tools like Vampirtrace [2] and Vampir [3][4], Paraver [5], TAU [6] are commonly using this approach, which is known as post-mortem analysis, to perform performance analysis on very-large systems with thousands of processors.
Instrumentation can be performed in two ways: either manually or automatically. Automatic instrumentation is done by using compiler techniques to introduce hooks at the enter and leave sections of each function without programmer intervention. The hooks are usually directed to a function to produce a log message in the trace file (see Fig. 1). With this technique one can analyze the time that the processor has been running inside each function or the number of times each function is called.

These tools give much richer information than a simple profiler because they add the time dynamics and therefore preserve the spatial and temporal behavior of the program run. Also, with proper instrumentation of communication functions, one can determine the communication pattern and overhead incurred by the application.

This is very useful, but there are several drawbacks mainly motivated by the instrumentation of the original application. When the application is instrumented, a small number of instructions are added to produce the log. Logs are usually written in memory first to minimize the time spent in slow disk access. Afterwards, when the analysis session is finished or the memory buffers have been filled, the logs are flushed to disk.

TABLE I. OVERHEAD FOR EMPTY FUNCTION CALL

<table>
<thead>
<tr>
<th>Instrumentation level</th>
<th>Instrumentation Overhead (in ticks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Manual</td>
</tr>
<tr>
<td>No instrumentation</td>
<td>15</td>
</tr>
<tr>
<td>Nothing</td>
<td>59</td>
</tr>
<tr>
<td>Function address</td>
<td>117</td>
</tr>
<tr>
<td>Function symbol</td>
<td>120</td>
</tr>
<tr>
<td>Function ID</td>
<td>119</td>
</tr>
<tr>
<td>Function ID + timestamp</td>
<td>229</td>
</tr>
</tbody>
</table>

With complex functions that are called at a low frequency the overhead introduced by the instrumentation is small compared with the time spent inside the function. However, for small functions the instrumentation time is comparable or even higher than the time inside the function and so the instrumentation entirely modifies the program behavior. This effect can be seen in TABLE I., which shows the overhead of an empty function call being instrumented manually, with the TAU source instrumenter called Program Database Toolkit (PDT) and with the -finstrument-functions flag of the GNU C compiler.

Modern Object Oriented (OO) languages encourage the programmer to use get/set methods, and pass the responsibility to the compiler to optimize the application performance by function inlining when possible. Using the function instrumentation provided by the compiler will then switch off all optimizations for these functions regardless of the chosen optimization level (Intel compiler version>10 and GNU compiler) which renders the tracing information useless. IBM and PGI compiler will favor inlining over instrumenting – with these compilers therefore all the trace information is not available at all. The usual alternative from the HPC community is to go to manual instrumentation. With manual instrumentation the programmer tells which functions he is interested in, while being careful to avoid tracing high frequently called functions. However when using this approach one can lose important information about the behavior of those non instrumented functions.

So the dilemma is to choose between 1) losing visibility by instrumenting just a subset of the functions and 2) analyzing a completely altered behavior because of the excessive instrumentation overhead.

III. WHOLE SYSTEM SIMULATOR

Could we have a third option that allows to fully instrument the code without adding overhead? Not in the real platform, but we could do that in a virtual prototyping platform by intercepting the function calls during execution and producing the traces in zero virtual time. Moreover, using this approach we could overcome the overwhelming problem of the storage demanded by logging. The problem of log storage is not an easy one; storage requirements for logging are also prohibitive for the HPC community. The embedded domain is characterized by the scarcity of resources, mainly memory. So having an alternative that is independent of the memory of the platform is a great deal.

Some virtual prototyping platforms are based on binary translation. In this kind of systems, since processor instructions are translated into the host platform instruction set, the execution time could be very different from the real execution platform. Although it would be perfectly possible, we do not advocate to use them for performance analysis purposes because their lack of time accuracy.

Instead we propose to use virtual platforms based on instruction set simulators. An instruction set simulator has time accuracy, and it can be modified easily to produce the traces when function call and return instructions are found during program execution. The benefit of this approach is that no hardware clocks are consumed when logging occurs and time accuracy, related to the execution time in the real platform, is preserved.
In order to simulate a complete MPSoC we have to instantiate several instruction set simulators, network interfaces and a NoC interconnect. As can be seen in Fig. 2, in this work we are using several instances of an Altera’s NIOS II instruction set simulator, which are connected to a NoC through a NIC circuit. The NIOS II instruction set simulator has been developed from scratch and it is described as HDL code at a mixed RTL and behavioral level. From an external perspective the ISS offers a CABA (Cycle Accurate, Bit Accurate) model. The behavioral model of each processor is triggered by the whole system simulator at every cycle. In fact, the ISSs are just like any other block of a big complex HDL design, they have inputs and outputs, and their execution is triggered by sensitive signals such as clock and reset.

The resulting system is a very detailed one with a very specific architecture. In the presented case (see Fig. 2) we have a cluster-based MPSoC with distributed memory. Obviously a shared memory architecture would have a very different block diagram, but, in essence, the ISS would have an equivalent role.

This particular architecture favors the adoption of a message passing programming model over other alternatives, such as shared memory programming models. Two main parallel programming models have been implemented and tested on many-core MPSoC architectures: OpenMP [7][8][9] (for shared memory architectures) and MPI [10][11] (for message passing architectures). Due to the inherent distributed nature of NoC-based many-core systems, we believe that, for many-core embedded systems, message passing architectures and programming models (e.g. MPI) could be mandatory to overcome the non-determinism and the scalability limits that cache coherence protocols introduce in shared memory architectures. Other reasons to support MPI vs. alternative APIs, like Multicore Association Communication API (MCAPI [12]), is that it is a very well-know API and parallel programming model, and debug and trace tools are currently available [4][5].

It comes as no surprise that we are using a lightweight version of MPI to implement applications on top of the presented architecture. Therefore, our compilation and execution process is slightly different to that used by the HPC community. Our MPI stack [15] is very lightweight and does not require any operating system, or any running daemon (i.e. mpirun). It is also remarkable that the program is not compiled once, but it is compiled several times to target each processor of the system. As depicted in Fig. 3, when creating multiple CPUs, the Altera’s SOPC builder creates a separate software library for each processor because each processor can be architecturally different. Differences could be related to cache sizes, FPU availability, attached devices or even the extension of the instruction set. To support this heterogeneity, the application source code must be compiled against each processor platform, thus producing different executables.

The compiled executables are executed by the ISSs that mimic the behavior of the Altera’s Nios II soft-core. The ISSs are incorporated together with the NICs and the NoC to provide a whole system simulation framework which is build on top of the NoCMaker simulation framework [14][19]. The outcome is an experimental environment that led us to create virtual Nios II NoC-based many-core systems.

This work improves other general purpose simulators, such as M5, Simics, and specially MPSoC simulators like MPARM [13], MCSIM [16], MRPSIM [18], ARTS [17] which do not include any tracing and performance analysis tools for next-generation many-core embedded and reconfigurable systems on the simulator.

We added support to the generation of Open Trace Format (OTF [22]) traces to our NoC-based MPSoC simulator at high-level on top of the parallel programming model [19], but also at instruction level whether custom message passing instructions are executed. This is possible because of the ability to extend the ISA of the base Nios II architecture.

The outcome is full insight into the virtual platform where the user can control the relationship between HW-SW components in order to improve HW-SW interfaces, SW stacks and parallel programming models [20][21] and the overall system performance of parallel applications.
IV. TRANSPARENT INSTRUMENTATION

As previously mentioned, the transparent instrumentation process is performed at the ISS level. This means that no code injection is performed at the compilation stage to insert the calls to logging functions. Instead, the ISS is slightly modified to do some additional actions when call and ret assembly instructions are found during program execution.

As depicted in Fig. 4, a call assembly instruction tells the processor to push the return address on the stack, and branch execution to the memory address where the called function is stored. We can obtain the called function name from this memory address and produce an enter function trace with the virtual time of the ISS. To obtain the function name from the address we need to incorporate in the system the information that is found in the executable memory map, which can be easily produced by the compiler during the linking phase. The virtual time can be also be easily obtained by looking at the number of clocks elapsed from simulation start.

When the ISS returns from the called function it executes the ret assembly instruction, which tells the processor to pop the return address from the stack and branch execution to that address. Since there is no direct way to identify what function the processor is returning from, we should keep track of the functions we are executing by using a stack. When a function is called its name should be pushed onto the stack so that when the processor leaves that function we can obtain its name and produce the corresponding leave message.

In this work we have created some functions to generate OTF files. OTF defines a format to representing traces for use with large-scale parallel systems. The OTF specification describes three types of files: a .otf file that defines the number of processors composing the system, a .def file which purpose is to define the different functions that we want to log in our system, and a number of .events files containing event specific data. OTF trace files can be be used by tools like Vampir [3][4], Paraver [5] or TAU [6]. All the results presented in this paper have been acquired with the Vampir tool. The trace format is composed of three fields: time field represents the relative time acquired when the instruction traced is executed, processor ID field shows the ID of the processor that executes the instruction, function ID field indicates the ID of the function traced, once a call or return instruction is detected the address of the instruction is compared with the memory-map, in order to obtain the function ID. Those traces will be logged each time the ISS enters or leaves the function that must be traced, which is when the ISS executes a call or a return instruction.

The memory-map list is obtained parsing the symbols table provided by the compilation of the program for a single processor. For each function an entry on the list is created provided by the compilation of the program for a single processor (see Fig. 5). The obtained list is used to generate the .def OTF file. The global memory-map list is necessary since it could be possible that a certain function could have a different ID value in each memory-map list of each processor of the system. The generation of this list guarantees that each memory map list has the same function ID for the same function name, ensuring in this way the coherence of the traces.

![Figure 4. Transparent Instrumentation in Instruction Set Simulator](image)

![Figure 5. Generation of the global .def file](image)
V. RESULTS

We use a Mandelbrot set calculation application to test our 2x2 NIOS II MPSoC Virtual Prototype. The application follows a typical Master-Slave parallel pattern, i.e. the master processor keeps sending and collecting parts of the workload that has to be computed until all the work is completed. In this case the workloads are very fine-grain since the master sends only the coordinates of a single point to each slave processor. The processor computes the Mandelbrot value of the received point and returns its computed value. The master processor keeps dispatching all the points to compute among the slave processors until all the work is completed.

In order to completely support our argument that automatically instrumentation and manual user instrumentation lead to erroneous performance analysis conclusions we could have compared automatically instrumentation, manual instrumentation and finally the transparent instrumentation we are proposing in this work. However, to implement automatic or manual instrumentation in our virtual prototype we would need to add a trace log memory to every processor, which would have to be large enough to store the logs, and whose contents should be saved to disk for later analysis. This would extraordinarily increase the complexity of our design to verify the well known fact that either with automatically or manual instrumentation the spatial and temporal behavior of the program might be altered. Instead, we test the Mandelbrot set calculation application in our original platform and see how the performance analysis is extremely useful to detect system bottlenecks and improve the performance of the system.

From previous experiments in a real synthesized FPGA-based platform we have observed that the current Mandelbrot set calculation program running on the 2x2 MPSoC does not get any speedup compared to a sequential version. In fact the application is severely slowed down when running in 4 processors.

So, the exact same application is run on the Virtual Prototype having parallel instrumentation to generate trace files, which we later analyze with Vampir. Vampir has lots of features but we can initially look at the Function Summary View to identify the functions where most time is spent. We have used a color code to make the function identification more easy, so orange and pink are associated to low level communication functions, light blue is associated to communication transport functions and dark blue is associated to the rest of the functions. Looking at this view (see Fig. 6) one can be misled to think that we should concentrate in optimizing nicCustomWrite and nicCustomRead functions to improve the performance and get some speedup.

But this would be completely wrong. If we look at the Timeline View and Process View from CPU0 (the master), we can see the time dynamics of this function calls. We should expect the master to be mainly sending coordinates and receiving results, but we easily realize that is spending a lot of time in transport functions. It is clear that no speedup will occur because no computation (dark blue areas) is overlapped among slave processors.

In order to optimize this particular application we decide to eliminate the communications transport layer and use alternative methods to work around some of their offered features (reception buffers, fragmentation and reassembly, etc.). The resulting software is compiled and tested again on the virtual prototyping platform. In Fig. 8 we can see that communication functions are no longer the dominant part, and that now, the computation of Mandelbrot values have become the most time consuming functions.
If we look at the Timeline View of the captured traces (see Fig. 9) we can easily see that computation is overlapped among slave nodes and that now the master node is sometimes idle waiting for the answer of the computing nodes.

When we go back to execute the application on the real synthesized platform on an FPGA we are able to exactly measure the benefit from this performance optimization. As shown in Fig. 10, the initial parallelization version was very poor because the excessive overhead of the communication transport layer was preventing any computation overlap of the worker slaves, and thus provoking an actual slowdown of the application from 18.64s to 52.74s (a 0.35x factor). However after the performance analysis and the corrective measures being taken the execution time goes down to 7.2s, which corresponds to a 2.59 speedup factor. This is a reasonably good result since the application has a very fine-grain partitioning of the workloads. Within an ideal system in which communication costs were negligible, we could expect a maximum speedup factor of 3, because, although we have 4 processors, the master node is only dispatching the workloads among the 3 worker slaves and not doing any computation.

VI. CONCLUSIONS

We have shown how some HPC performance analysis methodologies can be adapted and applied successfully to the embedded domain. The transparent instrumentation based on convenient Instruction Set Simulators has proven to be a viable and good solution for trace file generation, since it avoids the time behavior modification of the application that is usually observed when manual or automatic instrumentation is used. Moreover, using transparent instrumentation on ISS solves the problem of trace storage which is an overwhelming problem in the embedded domain.

A virtual prototype of a 2x2 NIOS II MPSoC has been built following this approach. It includes 4 independent NIOS II ISSs connected to a NoC through their respective NICs. The system is build over the NoCMaker simulation framework, which has been proven to be very adequate and flexible for this purpose.

The generation of OTF traces has allowed us to use the Vampir tool to analyze the performance of a Mandelbrot Set Calculation application, which was suffering from very poor performance. As a result, the application has been optimized and the execution time of the parallel code has been reduced from 52.74 to 7.2 seconds.

ACKNOWLEDGMENT

This work was partly supported by the European ITEA2 ParMA (Parallel programming for Multicore Architectures) Project (No 06015, June 2007 – May 2010) and by the Spanish Ministerio de Industria, Turismo y Comercio, Project Reference TSI-020400-2009-26 and Ministerio de Ciencia y Innovacion, Project Reference TEC2008-03835/TEC, the Catalan Government Grant Agency Ref. 2009SGR700 and the German Ministry for Research and Education (BMBF).
REFERENCES


