Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 736347, 14 pages
doi:10.1155/2012/736347

Research Article

Performance Analysis Techniques for Multi-Soft-Core and Many-Soft-Core Systems

David Castells-Rufas, Eduard Fernandez-Alonso, and Jordi Carrabina

CAIAC, Universitat Autònoma de Barcelona, Edifici Enginyeria, Campus UAB, 08193 Bellaterra, Spain

Correspondence should be addressed to David Castells-Rufas, david.castells@uab.cat

Received 9 March 2012; Revised 17 May 2012; Accepted 19 May 2012

Academic Editor: Patrick R. Schaumont

Multi-soft-core systems are a viable and interesting solution for embedded systems that need a particular tradeoff between performance, flexibility and development speed. As the growing capacity allows it, many-soft-cores are also expected to have relevance to future embedded systems. As a consequence, parallel programming methods and tools will be necessarily embraced as a part of the full system development process. Performance analysis is an important part of the development process for parallel applications. It is usually mandatory when you want to get a desired performance or to verify that the system is meeting some real-time constraints. One of the usual techniques used by the HPC community is the postmortem analysis of application traces. However, this is not easily transported to the embedded systems based on FPGA due to the resource limitations of the platforms. We propose several techniques and some hardware architectural support to be able to generate traces on multiprocessor systems based on FPGAs and use them to optimize the performance of the running applications.

1. Introduction

The emerging reconfigurable hardware devices allow the design of complex embedded systems combining soft-core processors with a mix of other IP cores. The reduced NRE costs compared to ASIC is a typical reason to choose FPGAs as the platform to implement some applications on [1]. However, the continuous increase of capacity and the flexibility offered by reconfigurable hardware are also important reasons to select FPGAs in order to get good time-to-market and time-in-market values. The actual FPGA strong demand let us speculate about the systems hosted by coming devices.

We predict that hundreds of soft-core processors will be hosted in future devices, giving birth to many-soft-core systems. This speculative claim is based on the observation of several trends. First, the capacity of integration has been dramatically growing during the last decades following Moore's law, and, in fact, already today, top-of-the-line devices are possible to host more than 100 simple soft-core processors (see Figure 1).

Second, companies are evolving EDA tools to make it easier to integrate a large number of processors. For instance, Altera recently introduced QSys [2], which can greatly simplify the tasks of designing tiled Noc-Based MPSoCs on a FPGA. Third, parallel programming is going mainstream in almost any computing platform. On multi-soft-core and many-soft-core processors parallel programming could be a faster way to improve performance before going to dedicated hardware, especially if users can reuse well-known programming frameworks like OpenMP and MPI. And finally, adding dedicated processors to take control of real-time tasks can be an easier option than integrating real-time operative systems (see [3]). So, the isolation of real-time tasks would help to maintain a demand for more processors.

In this context, software development becomes increasingly important for embedded systems. Performance analysis is a crucial phase of performance optimization but can also be an important part of the verification of the application of real-time constraints. This process involves several tasks (Figure 2). In summary, the idea is to collect information

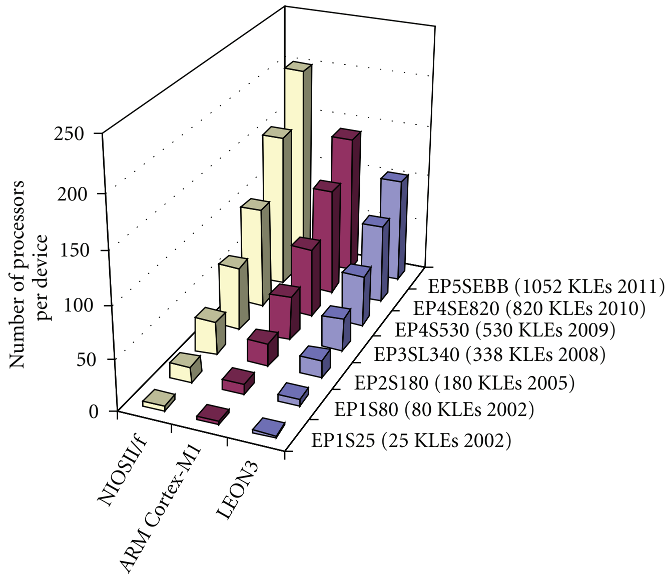


Figure 1: Evolution in the capacity of integration of Altera devices with respect to the theoretical number of soft-core processors that they might host.

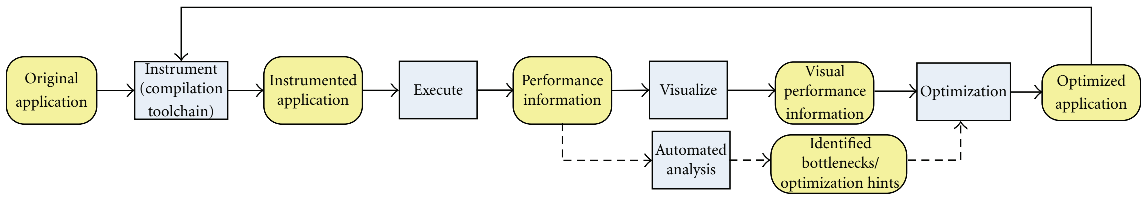


Figure 2: Performance of optimization process.

from different executions of the application to be later analyzed in order to provide some insight into the performance problems that can be solved in a next respin of the iterative coding process.

High-performance computing (HPC) systems have fewer problems than embedded systems to collect information and create traces for later analysis because they usually contain large amounts of memory. However, FPGA-based systems are much more resource-constrained, and specific strategies are necessary to provide equivalent features. In this paper, we propose strategies to make tracing possible for multiprocessors in FPGAs.

The rest of the paper is organized as follows. Section 2 describes various performance analysis techniques that have been used and proposed in the literature. Following sections describe various novel methods that we are proposing. In Section 3, we describe how automatic compiler instrumentation can be used in multi-soft-core systems with a low number of processors. In Section 4, we show a strategy to be used with distributed memory systems with a larger number of cores. An alternative way to capture event information with no overhead is presented in Section 5. Finally, in Section 6, a more complex approach combining specific hardware and an additional software transformation step are presented. We conclude with an overview of the techniques and their benefits and some mention to planned future work.

2. Previous Work

A large number of applications implemented on FPGAs described in academia focus on performance of the design related to other possible implementations. This is even the case for works based on soft-core processors. Surprisingly there is little work on more detailed performance analysis for soft-core systems. There are some very notable exceptions ([4, 5]) but do not focus on the specific multi-soft-core and many-soft-core specific issues.

Most works report total execution time (TET), which is a metric to compare the goodness of an implementation, and a way to choose between different implementations. Nevertheless, TET does not give the observability needed to understand why some expected behavior is not happening during execution. TET does not give any feedback on where the bottlenecks of the applications are. TET gives no information about the application dynamics in relation to the real-time constraints.

In the following subsections, we review some techniques that can be used in reconfigurable devices to shed some more light on the performance characteristics of the application under test.

2.1. Logic Analyzer-Based Analysis. A quite rudimentary approach to obtain time information is to assert a signal that

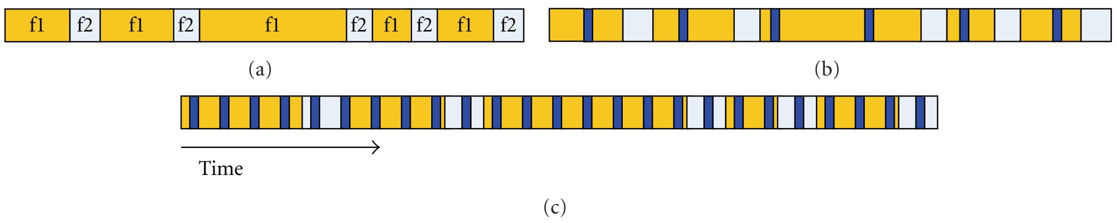


Figure 3: Tradeoffs between accuracy and overhead in Profiling. (a) Original application execution. (b) Profiling with low sampling frequency. (c) Profiling with high sampling frequency.

can be traced externally by a digital logic analyzer. This is a very old idea ([6]), but with the proliferation of embedded logic analyzers inside FPGA devices it became possible without the need of an external logging device ([7]). However, the limitation on the signals that can be traced, and the size of the trace buffer, does not make it a convenient way to analyze a typical complex application. On the other hand, if the application is partitioned in subsets that are analyzed separately by this technique, the number of resyntheses and run steps increases, making it an extremely time consuming process.

2.2. Profiling. Application profiling is usually understood as the process of gathering histogram information from the functions of an application. The usual observed metrics are the number of function calls and the time spent inside every function. Profilers can be based on periodically sampling and collecting the value of the program counter or either instrumenting the entry and exit of every function. Sampling profiling has usually lower overhead although it also has lower accuracy.

This tradeoff between accuracy and overhead is depicted in Figure 3. The blue boxes represent the time spent in the function that collects the necessary information. When sampling frequency is low (b), the original execution time of the application (a) is only slightly incremented, but the profiler never knows that function f2 is called. So the results will be misleading. On the other hand, if sampling frequency is incremented (c), the profiler will know that f2 is called but the execution time will be substantially affected.

The GNU toolchain includes the gprof profiler which is a sampling profiler. Since popular soft-cores use the GNU toolchain, it is also a known technique to get performance information from soft cores [8, 9]. It consists in a three-step process: first the application if instrumented by adding the `-pg` compilation flag in gcc. When the `-pg` flag is enabled, the compiler inserts a call to a function that will keep the count of every function invocation and registers a timer callback function that will record the program counter at every sampling period. Second, the application is executed and its execution generates an output file containing all the collected information. Third, the output file is analyzed by a visualization tool to present all the information to the developer. In the case of the Altera toolchain, this application is called `nios2-elf-gprof`.

2.3. Transparent Profiling. Sampling-based profiling can be very misleading if the sampling frequency is very low because there is a high probability to miss the execution of short functions. On the other hand, if the sampling frequency is very high, it can produce a high overhead causing a significant alteration on the usual application behavior.

An alternative approach to reduce the overhead is to monitor the program counter (PC) with specific additional hardware. Shannon and Chow in [10] described a method to continuously monitor the value of the PC. In that work, a hardware module includes a low address and high address register; comparison between logic and a counter. When the PC value is between the low and high value the counter is incremented. Those registers are typically programmed with the address boundaries of a function. In addition, the circuit can be replicated as many times as functions you have to measure. Although this approach is limited to just a small number of functions, the lack of overhead is appealing.

Profiling can expose some bottlenecks of the program, but it gives no details about the dynamics of the system and is not useful to identify whether the application meets the real-time constraints or not.

2.4. Trace-Based Analysis in Virtual Prototypes. In order to capture information of the time dynamics of an embedded application running on a multi-soft-core system, virtual prototypes can be used. In virtual prototypes we talk about the *host* processor, who runs the instruction set simulator, and the *target* processor, that is, the processor being simulated. A virtual prototype of such system would consist in several instruction-set simulators to model the multiple *target* soft-cores and some behavioral models to model the accompanying IP cores. In [11], we describe a system that provides a virtual prototype of a distributed memory multiprocessor containing 16 cores [12]. In this case, the ISSs are integrated as part of a HDL simulation framework that also hosts the HDL models of the rest of the system (see Figure 4).

In [11], we use transparent instrumentation, meaning that the ISS automatically logs the entering and exiting of every function call executed by the processor. Hence, the ISS source code has been modified to log the details of the execution of the *call* and *ret* machine instructions as shown in Figure 5.

The benefits of using transparent instrumentation in virtual prototypes are that we can overcome two of the typical issues faced in performance analysis. One, the potential excessive overhead caused by instrumentation. Here it is totally eliminated because we spend *host* time (instead of *target* time) to produce the traces. Two, we are not constrained

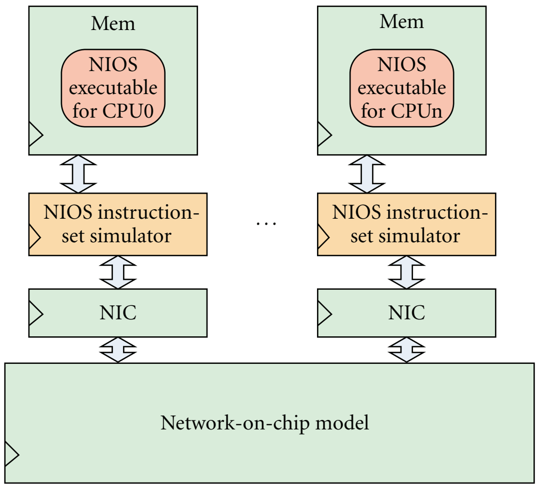


Figure 4: Logic design of a multi-soft-core virtual prototype.

by the limited memory or communication resources of the *target* system. So we can generate huge trace files into the host hard disk for later analysis.

Those trace files can be later visualized with tools like Vampir [13], and after human analysis the application bottlenecks can be found and solved to get new optimized versions of the applications. In the example depicted in Figure 6, we show how the visualization of the event traces shed light about the inefficiencies of the communication primitives used by implementation of a parallel version of the Mandelbrot application. The orange and pink color bars show the periods of time spent in communication libraries. In this case, the slave processors are wasting a lot of time in waiting for new messages, while the master processor is busy composing new messages through the communication stack (light blue color). Hence, no computation overlap is happening, and very low parallel efficiency is achieved. After the optimization of the communication primitives, computation is overlapping in slave processors, and a much higher parallel efficiency is achieved.

The concept of transparent instrumentation from the ISS can also be applied to obtain other kind of information from the processor. In [14], Hübert et al. describe *memtrace*, an extension to the ARM ISS ARMulator that allows capturing the memory accesses of an application.

2.5. Functional Simulation. A large body of research has been devoted to performance modeling based of high level of abstraction descriptions (e.g., [15–18]). Starting with RTL hardware simulation, one can try to speed up the system simulation by using higher-level models of the same circuits, or one can start directly from high-level design to get to the hardware implementation in an iterative process. This, of course, comes at losing accuracy (see Figure 7). Why is simulation preferred to execution in those works? The argument is that hardware design is a costly process, and it is desirable to start software integration as soon as possible to minimize the time-to-market. To do it, functional models can be quickly described to avoid postponing the development of software.

This makes sense for ASIC design and for the design of FPGA applications, where custom hardware must be developed from scratch, or when it is too expensive to replicate the hardware platform. However, for many FPGA-based systems the premise of the hardware not being available is more difficult to hold given the large number of IP Cores, the availability of soft-core processors, and the low cost of some FPGA platforms. So, in those cases, building a complex virtual platform does not make sense, since building the actual platform is easier, and, then, it is more convenient to measure timings directly actual platform to have full accuracy.

Virtual platforms can give a lot of information; however, they suffer several drawbacks. If simulation is done at a low level (RTL) or at a combination of levels (like in [11] or [15]) to include the existing descriptions of hardware IP Cores, the simulation speed is slow. Also, as mentioned before, if binary translation or functional simulation is used simulation is greatly accelerated but then time accuracy is lost, and some development effort is needed to provide IP cores as functional descriptions.

Moreover, to analyze the interaction with some real physical (nonvirtual) device and determine if the real-time constraints are met, virtual platforms cannot be generally used.

3. Automatic Compiler Instrumentation in Soft-Core Toolchains

To have some insights from the real execution platform, it is necessary to insert the tracing facilities in the FPGA device. Since most soft-core toolchains are based on gcc, they can benefit from the facilities provided to do automatic instrumentation of source code.

Automatic compiler instrumentation is activated by the `-finstrument-functions` gcc compilation flag. When this flag is present, the compiler automatically inserts a call to the functions `__cyg_profile_func_enter` and `__cyg_profile_func_exit` in the prolog and epilog of the called function as shown in Figure 8. Only the function prototypes of those functions are fixed: their implementation is free so they can be customized to fulfill any need as follows:

```
void__cyg_profile_func_enter (void *this_fn,
void *call_site);
void __cyg_profile_func_exit(void *this_fn, void
*call_site).
```

Nevertheless, since it is an "invasive" method that modifies the original application one should take care of avoiding, if possible, the instrumentation of functions that are called at a very high-frequency rate to avoid the problems depicted in Figure 3(c). This can be done by adding the `no_instrument_function` attribute to the high-frequency function prototypes. It is also recommendable to have a simple and fast implementation of enter and exit of the functions to minimize the overhead and try to maintain as much fidelity as possible with the original time constraints.

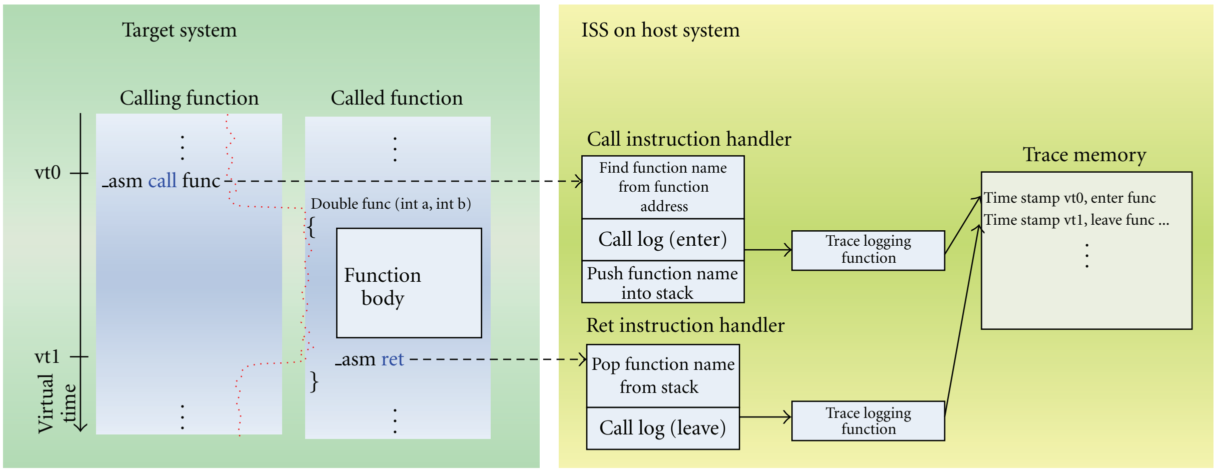


Figure 5: Transparent instrumentation method.

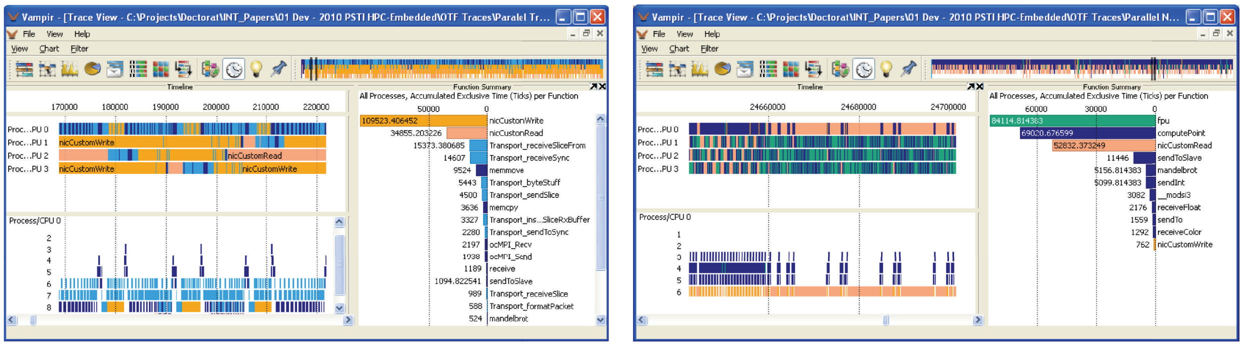

Vampir - [Trace View - C:\Projects\Doctorat\INT_Papers\01 Dev - 2010 PSTI HPC-Embedded\OTF Traces\Paralel Tr...
File View Help
View Chart Filter
Timeline
Proc...PU 0
Proc...PU 1
Proc...PU 2
Proc...PU 3
Process/CPU 0
Function Summary
All Processes, Accumulated Exclusive Time (Ticks) per Function
109523.406452 nicCustonWrite
34855.203226 nicCustonRead
15373.380685 Transport_receiveSliceFrom
14607 Transport_receiveSync
9524 memmove
5443 Transport_byteStuff
4500 Transport_sendSlice
3636 memcpy
3327 Transport_ins...SliceRxBuffer
2280 Transport_sendToSync
2197 ocMPI_Recv
1938 ocMPI_Send
1189 receive
1094.822541 sendToSlave
989 Transport_receiveSlice
588 Transport_formatPacket
524 mandelbrot
(a)
Vampir - [Trace View - C:\Projects\Doctorat\INT_Papers\01 Dev - 2010 PSTI HPC-Embedded\OTF Traces\Parallel N...
Function Summary
All Processes, Accumulated Exclusive Time (Ticks) per Function
84114.814383 fpu
69020.676599 computePoint
52832.373249 nicCustomRead
11446 sendToSlave
5156.814383 mandelbrot
5099.814383 sendInt
3082 __modsi3
2176 receiveFloat
1559 sendTo
1292 receiveColor
762 nicCustomWrite
(b)

Figure 6: Performance optimization process of a parallel Mandelbrot application. (a) Trace visualization of the initial version. (b) Trace visualization of the optimized version.

Simulation type	Speed	Accuracy
Functional execution	100000	
Timed native co-simulation	10000	
Timed binary translation	1000	
ISS (instructions)	100	
ISS (cycle accurate)	10	
Pin accurate	1	

Figure 7: Trade-off between speed and accuracy [17].

In our case, we present an implementation which is used in a multi-soft-core system containing 4 CPUs that control 4 motor/encoder pairs independently, and is executed in the Altera Cyclone IV low cost device included in the Terasic DE0-Nano board (see Figure 9). The logging approach is very straightforward, we save the function address into an array together with a time stamp obtained from a 64-bit counter. In complex systems like HPC machines, it is difficult to synchronize multiple clocks to act as a single clock for timestamping. On the contrary, in a multi-soft-core this becomes fairly simple due to the possibility to share a reset signal among all the counters.

The array containing the application logs is stored in the external SDRAM, which is limited to 32 MB. Some strategy has to be defined to avoid the logging of every call from the boot of the system. We define an internal flag to control when logging is enabled or disabled. During the development phase, we can enable this flag and capture all the events and later disable it to stop collecting logs. We can use it also to reduce the collected information for high-frequency functions.

Every CPU logs its activity independently, but when logs have to be transferred out to an external computer they are merged by the Master CPU (CPU0) and transmitted through a JTAG cable using the JTAG UART core provided by Altera. The format used in this communication is simple and does not follow any standard trace format, but once in the PC the traces are translated to the open trace format [19] to be visualized by standard trace analysis tools.

Using this strategy, the visibility of the application behavior is importantly increased. Figure 10 shows the visualization of some traces generated with this system in the Vampir tool. The top panel shows an overview of all the processors timeline, while the above four panels show the evolution of every call stack. Red color is used to identify

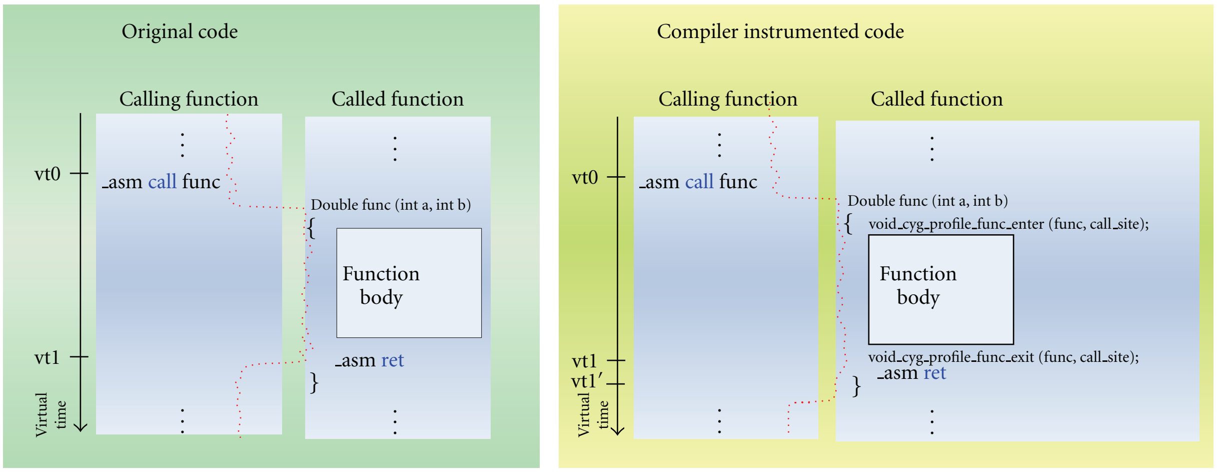


Figure 8: Gcc automatic compiler instrumentation.

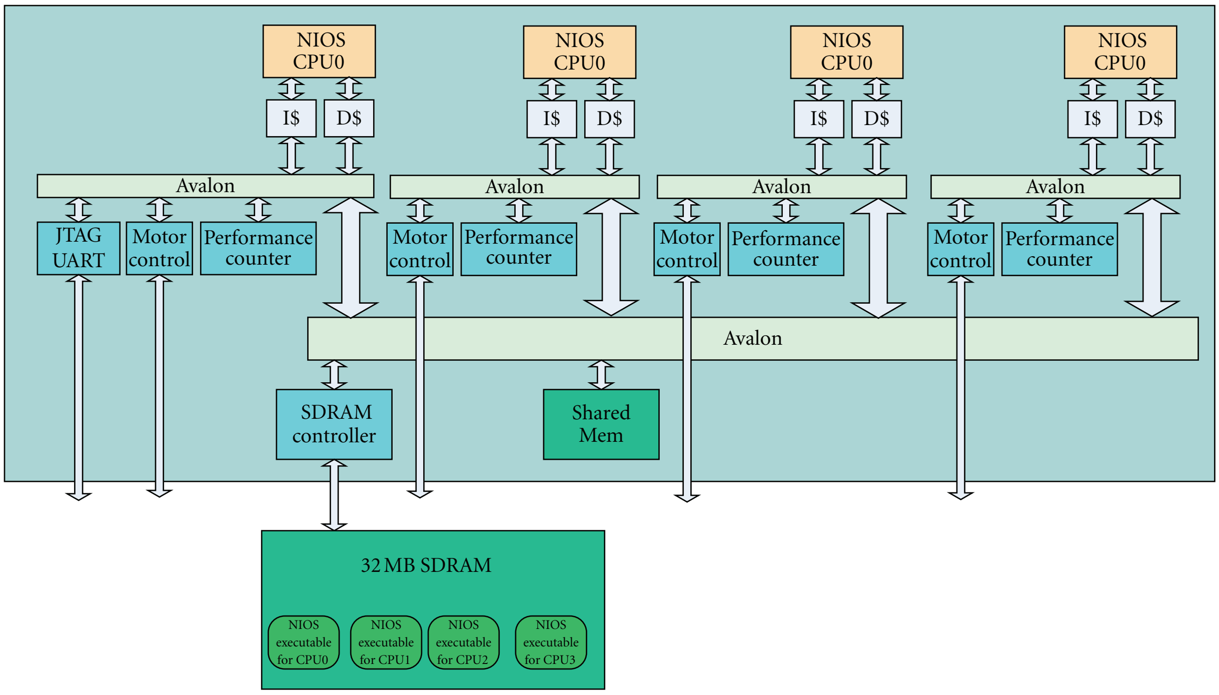


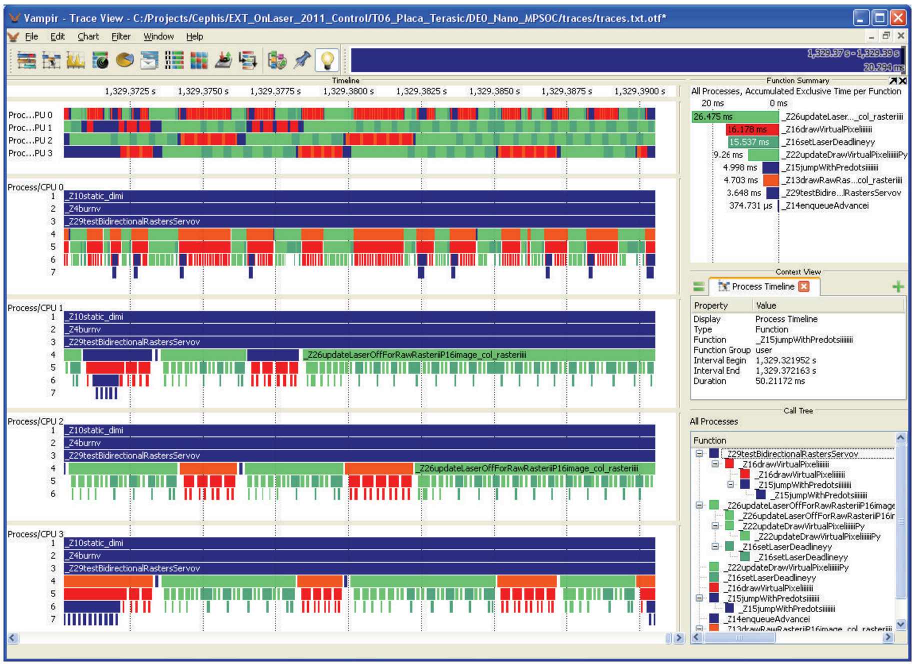
Figure 9: Multi-soft-core architecture on the Terasic DE0-Nano board.

the functions that work with a shared resource, which should be exclusively owned by a single CPU at a time. In the visualization, it is perfectly clear that the application is not behaving as expected as the red bars from several processors are overlapped. CPU1, CPU2, and CPU3 are perfectly coordinated to use the shared resource, but CPU0 is often using it when it should not. Moreover from the analysis of the duration of some functions, we can also deduce that CPU0 is working faster than the other CPUs. That could be caused by different clock signals feeding the CPU or by different dimensions of the processor caches.

This approach is very reasonable and allows a good visibility of the system with a controlled overhead. However, it is based on the existence of a shared memory than can store a large number of event information.

4. Dedicated Tracing Node

When designing large heterogeneous distributed memory many-soft-cores, writing logs to the global shared memory is not an option because, obviously, the architecture does not have a global common memory accessible from every core. In Figure 11, we show an architecture containing 16 NIOS processors that we have build in an Altera Stratix II S180 device. In this architecture, the 16 processors are interconnected by a network-on-chip (NoC), but only one CPU has access to



Function Summary

All Processes, Accumulated Exclusive Time per Function

20 ms	0 ms
26.475 ms	_Z26updateLaser..._col_rasteriiii
16.178 ms	_Z16drawVirtualPixeliiiiiii
15.537 ms	_Z16setLaserDeadlineyy
9.26 ms	_Z22updateDrawVirtualPixeliiiiiPy
4.998 ms	_Z15jumpWithPredotsiiiiiiii
4.703 ms	_Z13drawRawRas...col_rasteriiii
3.648 ms	_Z29testBidire...lRastersServov
374.731 μs	_Z14enqueueAdvancei

Property	Value
Display	Process Timeline
Type	Function
Function	_Z15jumpWithPredotsiiiiiiii
Function Group	user
Interval Begin	1,329.321952 s
Interval End	1,329.372163 s
Duration	50.21172 ms

All Processes

```
Function
⊟ _Z29testBidirectionalRastersServov
    ⊟ _Z16drawVirtualPixeliiiiiii
        _Z16drawVirtualPixeliiiiiii
      ⊟ _Z15jumpWithPredotsiiiiiiii
          _Z15jumpWithPredotsiiiiiiii
⊟ _Z26updateLaserOffForRawRasteriiP16image
    _Z26updateLaserOffForRawRasteriiP16ir
  ⊟ _Z22updateDrawVirtualPixeliiiiiPy
      _Z22updateDrawVirtualPixeliiiiiPy
  ⊟ _Z16setLaserDeadlineyy
      _Z16setLaserDeadlineyy
  _Z22updateDrawVirtualPixeliiiiiPy
  _Z16setLaserDeadlineyy
  _Z16drawVirtualPixeliiiiiii
⊟ _Z15jumpWithPredotsiiiiiiii
    _Z15jumpWithPredotsiiiiiiii
  _Z14enqueueAdvancei
⊟ _Z13drawRawRasteriiP16image_col_rasteriiii
```

FIGURE 10: Trace-based analysis using Vampir of a compiler-based generated trace.

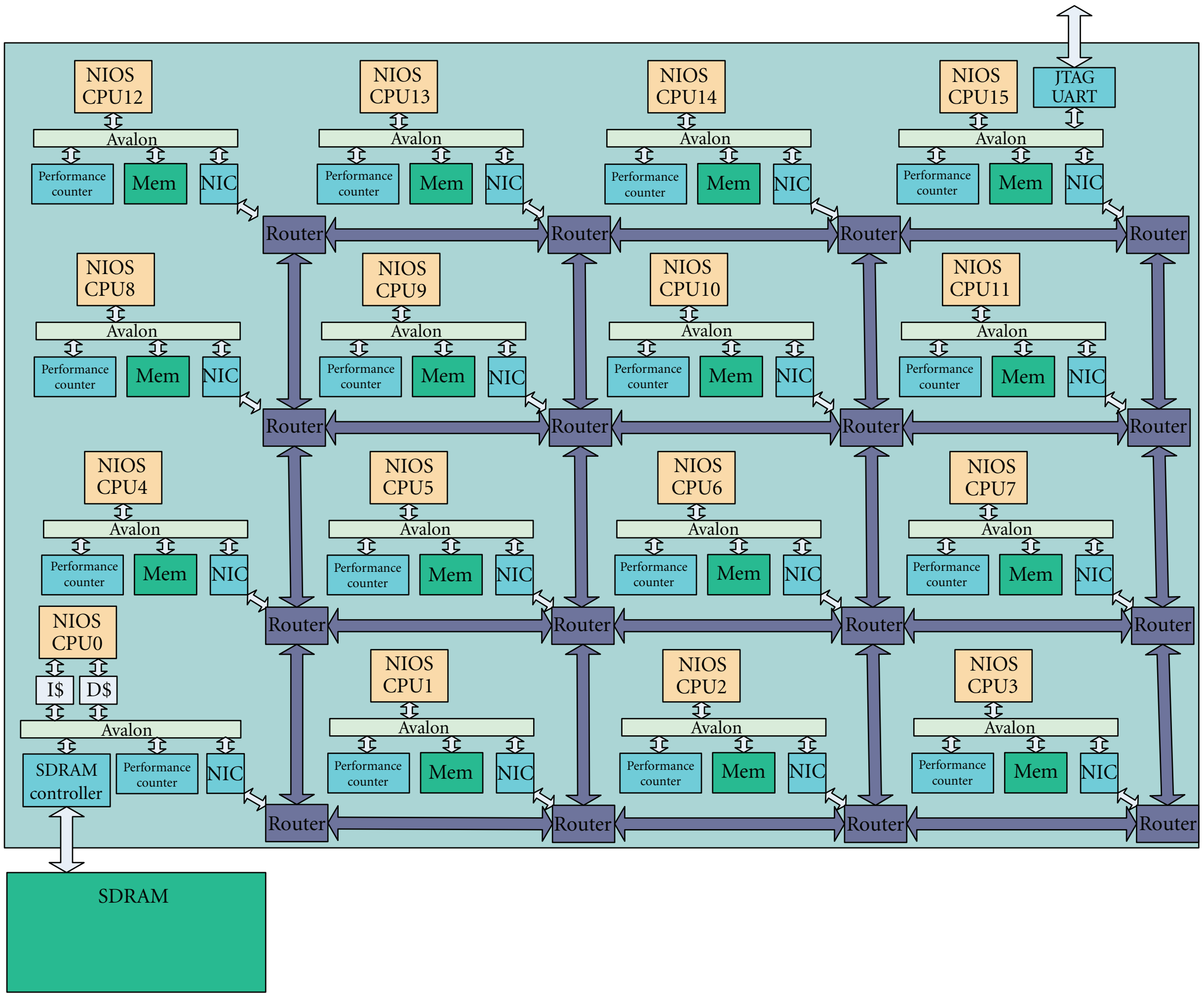


FIGURE 11: Large distributed memory architecture many-soft-core.

the big external SDRAM memory, while the rest of the processors have small on-chip memories. Those small memories are big enough to host some application kernel code plus the input and output data of their computation before it is transferred out to another processor. However, there is no option to use those memories to store logs of event information.

To solve this limitation, we have devoted one of the processors to generate the traces and store them temporally before sending them to an external computer for visualization. All the other processors have to send the event information through the NoC to the tracing node.

The drawback of this solution is that the same NoC is used by the original application and the tracing infrastructure. In addition, the injection of packets to the network has a bigger overhead than writing to a shared memory. Even a more serious problem is that if events are generated at high-frequency rate, the network can saturate affecting the performance of the whole system and potentially causing a significant modification of the usual behavior an even causing a false failure to meet the real-time constraints of the application.

5. Transparent Instrumentation

A partial solution to the alteration of the normal behavior of the system when high-frequency events occur is to try to separate the execution of the application being analyzed from the generation of the traces in different processors and pretend that time "freezes" in the application under analysis when an event is generated.

This idea shares the concept of having two different times with the transparent instrumentation approach we do in virtual prototypes. However, the translation in hardware of the idea needs to make use of clock gating. So, when an event occurs, some entity has to stop the clock of the processor and produce the trace. Since we need a synchronized global clock, in fact, we have to stop all the processors when any event happens in any of the processors until the event is correspondingly traced. Moreover, we should consider that events can happen simultaneously, so the entity handling the events has to support the simultaneous activation of several processors.

One difficulty we have faced in testing this idea is that we must monitor some internal signals from the processor Pipeline. To be more specific, we need to monitor the value of the PC and the signals produced after the Instruction Decoding pipeline stage that are asserted when a *call* instruction or a *ret* instruction are executed. Since those signals are usually hidden from the user in commercial soft cores, we have developed our own replica of NIOS processor from scratch (called MIOS) to be able to export *execCall*, *execRet,* and the PC signals. Those signals are fed to a trace unit that immediately gate the clock of all processors and handles the event information to send through the JTAG UART device, as shown in Figure 12.

In our design, the implementation of the trace unit is based on another MIOS processor. In order to minimize resources, the processor uses only 512 Bytes of memory, which are enough to host the data and the code (which has been written in assembly language) to process the event information and send it to the external computer. Table 1 shows the synthesis results for the whole tracing unit module, that consist on the processor, the memory, the JTAG UART, and the modules that handle the detection of events and the generation of the clock gating signal.

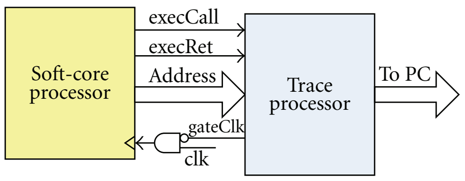


Figure 12: Hardware transparent instrumentation.

Table 1: Synthesis results for the tracing unit on the Ultera Stratix II S180 device.

Logic usage	3443 LCs
Memory usage	4096 bits
$f_{\max}$	114 MHz

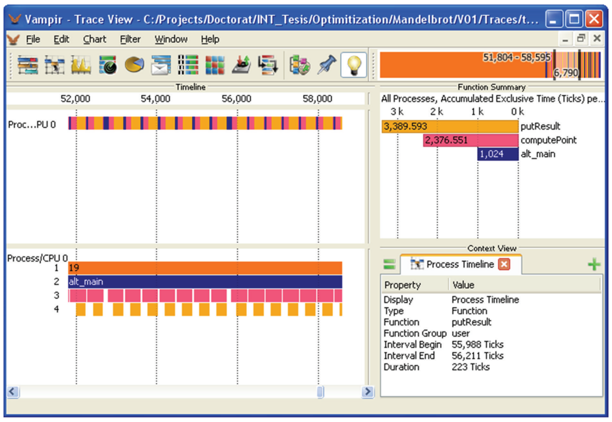
With this approach, it is perfectly possible to analyze high-frequency functions with no overhead. Figure 13 depicts how it is used to optimize a Mandelbrot application. In the left panel we see how a lot of time is spent in `putResult` function, which is colored in orange. The real computation takes place in `computePoint` function, and `putResult` is just a helping function to calculate what array index the computed point should be stored in. After some analysis, we realize that with some refactoring the array index can be obtained without complex computation. Indeed, the optimized version reduces significantly the time spent in `computePoint` (in pink color) and finally doubles the performance.

In transparent instrumentation based on global clock gating, we can use the concept of the virtual time, that is, the time that can be sensed by the clock that is gated. Virtual time is a time that we can stop. Unfortunately, real time cannot be stopped. The real time continues to run when the virtual time is stopped, so applications with real-time constraints will sense a different time behavior when using transparent instrumentation. In this approach, the more the number of processors or the frequency of function calls increases, the more the virtual time differs from reality. Obviously, this is an undesired effect that can be only minimized reducing the frequency of events that stop the virtual clock, which poses a contradictory demand to our initial goal of being able to sample high-frequency events.

Another drawback of this approach is the lack of support of commercial soft-core processors to export the necessary internal signals that are fundamental to implement such system.

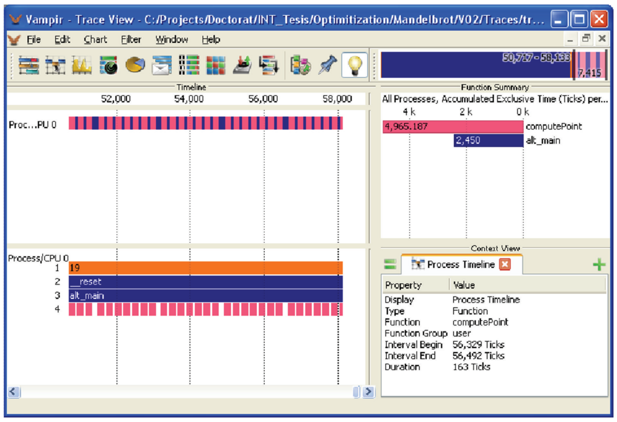
Table 2: Operations implemented by the NIOS custom instruction devoted to tracing.

Operand A (type of operation)	Operand B (address)	Operation description
00	—	Enables logging
01	—	Disables logging
02	Function address	Sends the address of a call
03	Function address	Sends the address of a ret



Property	Value
Display	Process Timeline
Type	Function
Function	putResult
Function Group	user
Interval Begin	55,988 Ticks
Interval End	56,211 Ticks
Duration	223 Ticks

(a)


Vampir - Trace View - C:/Projects/Doctorat/INT_Tesis/Optimitization/Mandelbrot/V02/Traces/tr...
File Edit Chart Filter Window Help
7,415
Timeline
52,000
54,000
56,000
58,000
Proc...PU 0
Function Summary
All Processes, Accumulated Exclusive Time (Ticks) per...
4 k
2 k
0 k
4,965.187
computePoint
2,450
alt_main
Process/CPU 0
19
__reset
alt_main
Context View
Process Timeline

Property	Value
Display	Process Timeline
Type	Function
Function	computePoint
Function Group	user
Interval Begin	56,329 Ticks
Interval End	56,492 Ticks
Duration	163 Ticks

(b)

Figure 13: Performance optimization of Mandelbrot application using transparent tracing. (a) Trace visualization of the the original version. (b) Trace visualization of the optimized version.

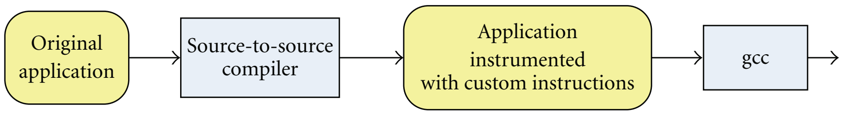

Original application
Source-to-source compiler
Application instrumented with custom instructions
gcc

Figure 14: modified toolchain to include the source to source compiler that produces the custom instructions that generate function traces.

6. Custom Instructions and Dedicated Networks

As we have been insisting several times, collecting the information to produce a trace needs time and causes overhead. As we detailed in [11], a simple instrumentation can already take few hundreds of clock cycles. Those cycles are devoted to function entering and leaving, stack manipulation, and memory operations to store the collected information. The transparent instrumentation can mitigate this overhead but needs a direct access to the some of the internal processor pipeline signals which are not usually available in commercial soft-core processors. An alternative way to generate event information with minimum overhead is to use custom instructions (in Altera NIOS) or FSL channels (in Xilinx MicroBlaze). By using custom instructions (CIs), we eliminate the clock cycles devoted to function entering, function exiting, stack manipulation, and even memory operations. In fact, the invocation of custom instructions can take just a single clock cycle, which is a minimal overhead suitable for most applications.

Since gcc does not natively support the possibility to instrument the functions with custom instructions, we have created a source-to-source compiler (instrument-custom) based on the Mercurium framework ([20]) to be interposed on the NIOS toolchain (as depicted in Figure 14).

The *instrument-custom* compiler translates the original source code into a modified code that includes the necessary macros to emit the custom instructions created for this purpose. An illustrating example of the result is shown in Figure 18.

The created custom instructions are implemented as a simple, very few clocks latency operation, which takes two parameters, the first is the type of operation, and the second is the address of the function. Table 2 describes all the operations implemented by the custom instruction.

When the custom instruction is executed, the generated event information has to be immediately sent to a remote node that is responsible to collect the traces. But to avoid the problems of congestion and saturation we previously described in Section 4, we need to create an additional network specialized in transferring the event info. The requirements of this network are very simple, if we face the worst case, we need n unidirectional channels from each processor to the tracing unit, which has to be able to eject the information coming from all the channels simultaneously. As shown in Figure 15, the network can be built just based on registers

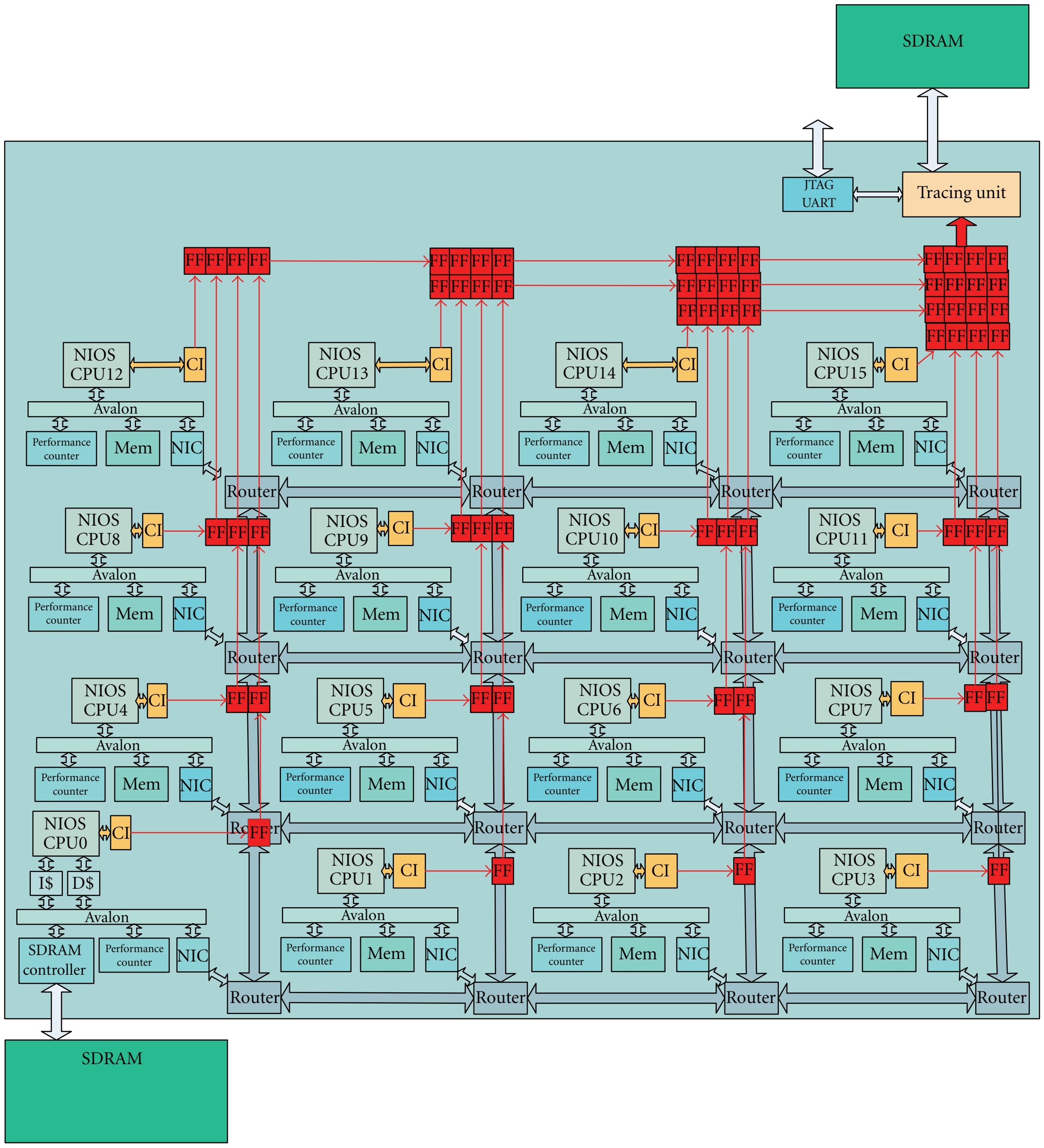


Figure 15: Complex trace logging architecture.

that are forwarding the event info to the tracing unit. A good point of this design is that it is not necessary to send the timestamps together with the address because they can be deterministically deduced in the tracing unit by the number of hops of every channel. For instance, CPU0 is at 7 hops from the tracing unit, so all the events coming from that channel will be timestamped by T-7. The total number of registers in the network is 2048.

The tracing unit has to eject 512 bits at every clock cycle to filter out the uninteresting information and manage the interesting one. In some FPGA boards, the information could be stored to a secondary memory port exclusively used for tracing. This is possible, for instance, in Altera Stratix IV development kit, that include two DDR3-independent memory banks.

Even though we designed the system to support one event per cycle from each processor, this scenario is not a realistic one. A more reasonable worst-case scenario (WCS) would be the result of having very high-frequency functions. In such a case the probability of events in each individual processor

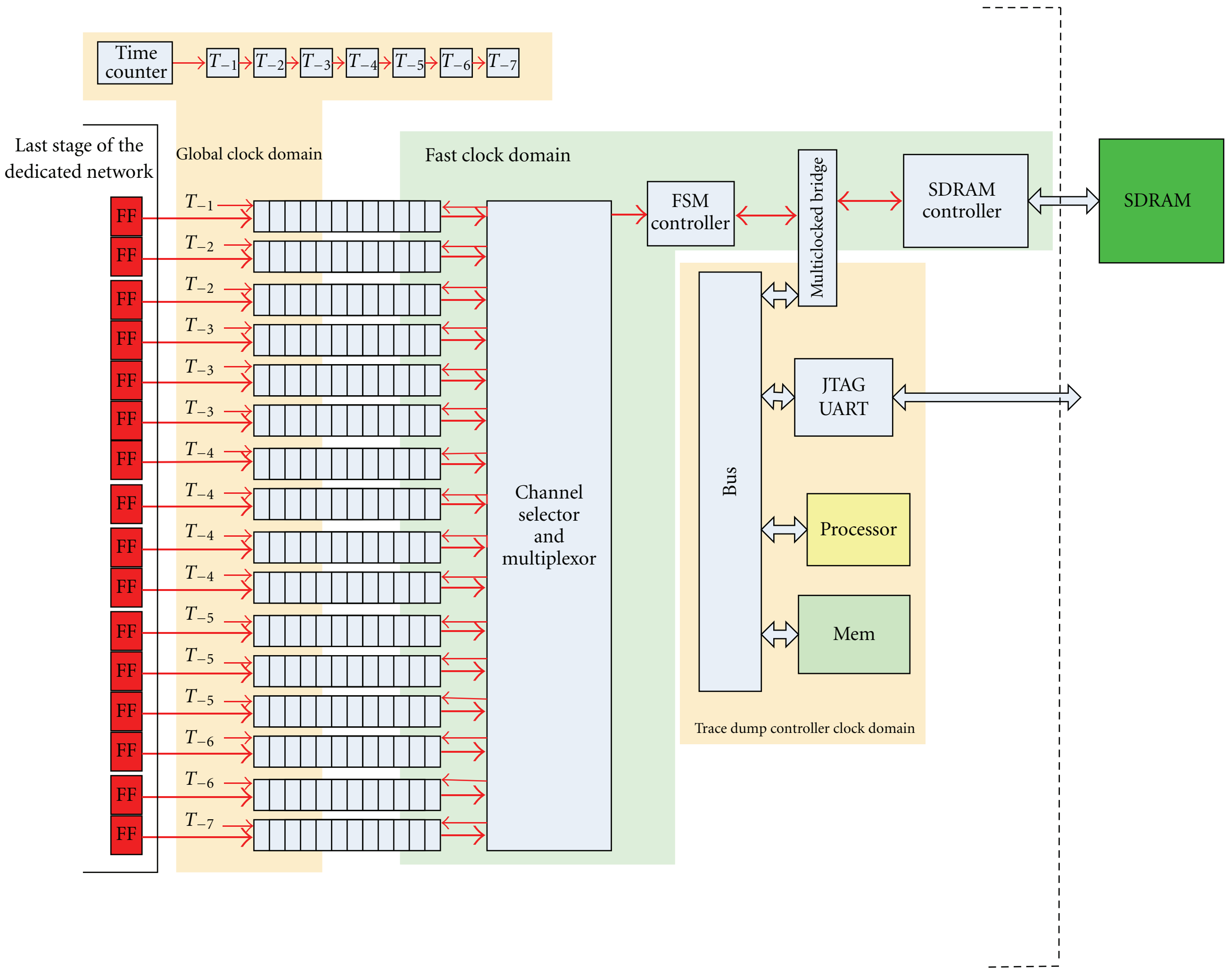


Figure 16: Architecture of the tracing unit.

is still below 1/10. Meaning that between the execution of the call and the ret instruction of a high-frequency function, there are almost always more than 10 cycles.

The injection capacity of the dedicated network is 512 bits per clock (32 bits per event per 16 nodes), but the processors very seldom use that capacity, and when they do not do it they inject a null value to the network meaning no event occured. After filtering out the no-events from the network, the actual WCS rate of output at the other endpoint of the dedicated network is less than 64 bits per clock.

We propose a tracing unit architecture as depicted in Figure 16. The event information coming from the network is combined with the timestamp associated for each channel, which is derived from a global timestamp plus a deterministic value depending on the number of hops to the other endpoint of the channel. Those pairs are feed to FIFOs that are needed to handle the potential collisions. A collision could be produced, for instance, when several (or even all) processors execute a *call* instruction at the very same clock cycle. Then a channel selector is needed to be able to select events from the FIFOs, that are not empty and transferred to the next module that will be responsible to communicate with the SDRAM controller for its writing in the external memory.

The external memory must be also accessible from another processor so that the collected traces can be exported out to a PC for their analysis.

The benefits of this approach is that it is more scalable than all the previous ones, although the number of processors increase the risk of saturating the SDRAM controller increases, and then rising similar problems exhibited by the previous approaches. Since the network is composed by totally independent unidirectional channels, the system can be easily fragmented so that it can flush the events of a subset of processors in different memory banks. That would need several tracing units collecting the events from different subsets of processors and storing them in different memory banks. Eventually, the limit to scalability would be driven by the number of memory banks that the FPGA can address, which is mainly a function of the number of pins of the FPGA device.

Table 3: Performance analysis methods that can be used in multi-soft-core and many-soft-core applications.

Acronym	Technique	Area overhead	Virtual time overhead	Real time overhead	Real time accuracy	Scalability
ELA	Embedded logic analyzer [7]	Medium	None	None	Full	Very low
SP	Sampling profiling [8, 9]	None	Low-high	Low-high	High-low	High
TP	Transparent profiling [10]	Medium	None	None	Full	Low
VP	Trace based on virtual prototypes [11]	—	None	High	Very low	Low
FS	Functional simulation [15–18]	—	—	Low	Very low	High
			New proposed methods			
ACI	Automatic compiler instrumentation	None	Low-high	Low-high	High-low	Medium
TN	Dedicated tracing node	Low	Low-high	Low-high	High-low	Medium
TI	Transparent instrumentation with clock gating	Medium	None	Medium	Medium	Medium
CI	Custom instruction and dedicated networks	Medium	Very low	Very low	Very high	High

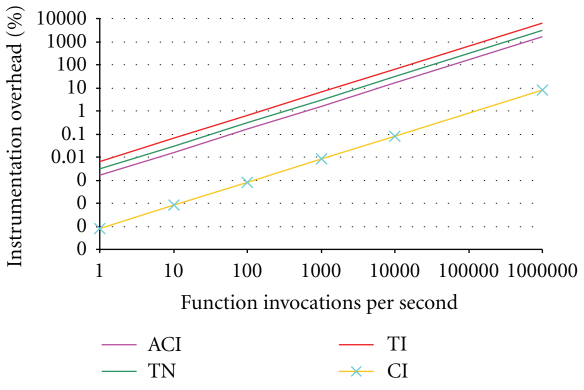


Figure 17: Instrumentation overhead as a function of function invocations per second for the different proposed methods.

7. Discussion

In the previous sections, we have analyzed some performance analysis methods that can be applied to multi-soft-cores and many-soft-cores. We started by describing the ones found in the literature to propose additional ones. Table 3 summarizes all presented techniques and the qualitative values they have in properties like area overhead, time overhead, real-time accuracy, and scalability. By area overhead we mean the amount of additional resources of the FPGA devoted to the measurement system. This concept is not applicable to virtual prototypes and functional simulation because no real platform is used. With time overhead, we make a distinction between virtual time and real time. Virtual time overhead is the amount of extra processor time devoted to measurement. In the approaches that use transparent instrumentation, this value will be zero because, from the processor perspective, no time is consumed by the measurement system. On the other hand, we use real-time overhead to denote the overhead measured from the wall clock. Another reported property is the real-time accuracy. A high-accuracy value means that the time characteristics of the original application are very similar to the time characteristics exhibited by the application when the measurement system is in place. A low accuracy will mean that the timing characteristics are totally different from the original application. Finally, by scalability we denote the capacity to extend the measurement system to work for a larger number of processors. A low scalability value can be caused by a prohibiting number of additional hardware resources, or by an increasing time overhead.

In some cases, we report a undetermined value going from low to high. We do that to show that the property can take different values depending on the value of other properties. For instance, in sampling profiling technique the real-time accuracy can be high if the sampling period is low and the application has low frequency functions, but if sampling period is high, then the accuracy is lowered.

In most methods, the real-time accuracy is dependent on properties like function frequency or sampling period. ELA, TP, and CI have very high accuracy, but the first two (ELA and TP) provide poor information and are not very scalable.

For many-soft-core systems, we need an scalable solution that can go up to hundreds of cores. Most methods have a limited scalability except FS and CI.

To have a better understanding of the source of the penalties incurred by each method, we provide (see Table 4) quantitative results of extra the area overhead expressed as number of hardware resources (LUTs and FFs), and the overhead affecting the real execution time expressed as number of cycles per called function.

Depending on the number of function invocations per second on the executing application, the real time overhead becomes more or less significant. We might consider that a good metric to compare the methods could be the time overhead added by trace generation expressed as a percentage of the original execution time. In Figure 17, we show the percentage of overhead as a function of the number of function invocations per second. The use of custom instructions outperforms all the other methods by several orders of magnitude while allowing to capture the traces from applications with more than 100,000 function invocations at a minimum overhead of 1%.

Table 4: Absolute area and time overhead of different methods.

Acronym	Technique	Area overhead	Real-time overhead
ACI	Automatic compiler instrumentation	0	793 cycles per function call
TN	Dedicated tracing node	324 LUTs + 1200 FFs	1532 cycles per function call
TI	Transparent instrumentation with clock gating	2459 LUTs + 1815 FFs	3216 cycles per function call
CI	Custom instruction and dedicated networks	13944 LUTs + 33614 FFs	4 cycles per function call

Original code

```
void putResult(float  x0, float  y0, int v)
{
   float  incx = (xmax-xmin) / divx;
   float  incy = (ymax-ymin) / divy;
   int x = ((x0 - xmin) / incx);
   int y = ((y0 - ymin) / incy);

   int index = y*((int)divx)+x;

   results[index] = v;
}
```

Transformed code

```
void putResult(float  x0, float  y0, int v)
{
   __builtin_custom_inii(0, 2 , putResult);

   float  incx = (xmax-xmin) / divx;
   float  incy = (ymax-ymin) / divy;
   int x = ((x0 - xmin) / incx);
   int y = ((y0 - ymin) / incy);

   int index = y*((int)divx)+x;

   results[index] = v;

   __builtin_custom_inii(0, 3 , putResult);
}
```

Figure 18: Example of the transformation performed by the source to source compilation phase. The left code is the code before the transformation, and the right code is the result of the transformation phase. Notice how the custom instructions have been inserted appropriately.

8. Conclusions and Future Work

Performance analysis is an important part of the optimization of parallel applications. The advent of multi-soft-core and many-soft-core architectures will make evident that there is a need to extract performance information in a systematic way.

We have reviewed various techniques used in previous works and proposed various novel techniques that can be applied to future many-soft-core systems. Automatic compiler instrumentation can be an excellent solution for multi-soft-core systems that can afford an external shared memory. Because of this reason, it is perceived as a non-scalable solution, and its overhead is too high for high-frequency functions. To overcome the first problem of scalability, the NoC can be used to transfer the event information, but then the system can suffer from network saturation or can be penalized by the application existing traffic. To overcome the second problem, transparent profiling based on clock gating can be used, but then real-time constraints can be missed. Eventually, we propose architecture based on custom instructions, a dedicated unidirectional event network and a specific tracing unit. The custom instruction provides a very low overhead (very few clock cycles) to generate the event log. The dedicated network is designed so that no congestion or saturation can occur. Finally, the tracing unit is designed to be able to store the data and export it to an external computer.

In future work, we will try to test this approach for many-soft-core systems containing 100 processors.

Acknowledgments

This work was partly supported by the European cooperative CATRENE project CA104 COBRA, the ITEA2 project 10021 MANY, the Spanish Ministerio de Industria, Turismo y Comercio projects TSI-020400-2010-82, and TSI-020100-2010-1036, and MEC TEC2008-03835; the Catalan Government Grant Agency Ref. 2009SGR700.

References

[1] P. H. W. Leong, "Recent trends in FPGA architectures and applications," in *Proceedings of the 4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA '08)*, pp. 137–141, January 2008.

[2] Altera, Qsys System Integration Tool, http://www.altera.com/products/software/quartus-ii/subscrip.

[3] P. E. McKenney and D. Sarma, "Hard real-time response," Patent US, 7748003, 2010, http://www.google.com/patents/US7748003.

[4] J. Curreri, S. Koehler, A. George, B. Holland, and R. Garcia, "Performance analysis framework for high-level language applications in reconfigurable computing," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, article 5, 2010.

[5] S. Koehler, G. Stitt, and A. D. George, "Platform-aware bottleneck detection for reconfigurable computing applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 3, article 30, 2011.

[6] T. S. J. Sun and R. D. Leu, "Software performance analysis using hardware analyzer," Patent US, 5903759, 1999, http://ip.com/patfam/xx/25351944.

[7] Altera Corporation, "Design Debugging Using the SignalTap II Embedded Logic Analyzer," 2007, http://www.altera.com/literature/hb/qts/qts_qii53009.pdf.

[8] Altera Corporation, "Profiling Nios II Systems," 2005, http://www.altera.com/literature/an/an391.pdf.

[9] J. G. Tong and M. A. S. Khalid, "A comparison of profiling tools for FPGA-based embedded systems," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECD '07)*, pp. 1687–1690, April 2007.

[10] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/software codesign," in *Proceedings of the 12th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, pp. 190–199, February 2004.

[11] D. Castells-Rufas, J. Joven, S. Risueño et al., "MPSoC performance analysis with virtual prototyping platforms," in *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW '10)*, pp. 154–160, September 2010.

[12] E. Fernandez-Alonso, D. Castells-Rufas, S. Risueño, J. Carrabina, and J. Joven, "A NoC-based multi-softcore with 16 cores," in *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems (ICECS '10)*, pp. 259–262, December 2010.

[13] A. Knüpfer, H. Brunst, J. Doleschal et al., *The Vampir Performance Analysis Tool-Set Tools for High Performance Computing*, Springer, 2008.

[14] H. Hübert, B. Stabernack, and K.I. Wels, "Performance and memory profiling for embedded system design," in *Proceedings of the IEEE 2nd International Symposium on Industrial Embedded Systems (SIES '07)*, pp. 94–101, July 2007.

[15] M. Montón, A. Portero, M. Moreno, B. Martínez, and J. Carrabina, "Mixed SW/systemC SoC emulation framework," in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE '07)*, pp. 2338–2341, June 2007.

[16] H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco, "System-level performance analysis in systemc," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, pp. 378–383, February 2004.

[17] H. Posadas, S. Real, and E. Villar, *M3-SCoPE: Performance Modeling of Multi-Processor Embedded Systems for Fast Design Space Exploration Multi-Objective Design Space Exploration of Multiprocessor SOC Architectures: The Multicube Approach*, vol. 19, Springer, 2011.

[18] I. Böhm, B. Franke, and N. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in *Proceedings of the 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS '10)*, pp. 1–10, July 2010.

[19] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (OTF)," in *Proceedings of the 6th International Conference, Part II, Computational Science (ICCS '06)*, N. Vassil Alexandrov, G. D. van Albada, M. A. Peter Sloot, and J. Dongarra, Eds., vol. 3992, pp. 526–533, Springer, Reading, UK, May 2006.

[20] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: a research compiler for openmp," in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004.