Runtime Multitasking Support on Reconfigurable Accelerators

Mojtaba Sabeghi, Hamid Mushtaq, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{M.Sabeghi, H.Mushtaq, K.L.M.Bertels}@tudelft.nl

ABSTRACT

Serving several applications at runtime on a reconfigurable machine is a challenging problem in which the reconfigurable fabric has to be shared among competing tasks. Due to the inherent complexity of assigning tasks to the FPGA, a comprehensive runtime system is required to address all the conflicting issues between competing applications' demands and to keep the system performance at the required level. In this paper, we present a runtime environment wherein a number of components introduced to handle the task assignment problem in a very low overhead fashion. We present the details of each component and evaluate the overhead imposed by each.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management – *Multiprocessing/multiprogramming/multitasking, Scheduling*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Reconfigurable Systems, Multitasking, Runtime Support.

1. INTRODUCTION

Commodity desktops and servers are moving towards employing reconfigurable fabrics in order to achieve higher performance. In such general-purpose reconfigurable computers, serving multiple concurrent applications is an evident need which imposes utilizing reconfiguration aware resource management techniques. In contrast to a majority of the current research approaches that the compiler deals with the FPGA resource management at compile time, in such systems due to lack of information about the runtime load, the compiler cannot assign the resources. Most of the current trends have assumed only a single thread of execution where the given application has full ownership of both the host microprocessor and the reconfigurable processors. Even in case of multiple applications, the scheduling is static as they assume the designer knows everything about the applications execution and the system load at design time. To have a more generalized and flexible system, this paper presents a run-time environment, which is responsible to fully operate the system and address all the conflicting issues between competing applications' demands.

This work was presented in part at the first international workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2010), Tsukuba, Ibaraki, Japan, June 1, 2010.

The rest of the paper is organized as follows. In section 2 and 3, we provide a summary over the related works and the background information. Section 4 presents our proposed profiler followed by the proposed scheduling algorithms in section 5. We describe our dynamic binding mechanism in section 6 and conclude the paper in section 7.

2. RELATED WORK

The IBM Lime [1] goal is to create a single unified programming language and environment that allows (all) portions of a system to move fluidly between hardware and software, dynamically and adaptively. Lime targets Java applications to be dynamically translated for co-execution on general-purpose processors and reconfigurable logic. Another similar work is PPL [2] which tries to extend the java virtual machine approach by featuring a parallel object language to be executed on a common parallel runtime system, mapping this language onto the respective computing nodes.

ReconOS [3] aims at the investigation and development of a programming and execution model for dynamically reconfigurable hardware devices. ReconOS extends the concept of multithreaded programming to reconfigurable logic. Another comparable work is BORPH [4], which introduces the concept of hardware process that behaves just like a normal user program except that it is a hardware design running on a FPGA. Hardware processes behave like normal software programs. The BORPH kernel provides standard system services, such as file system access, to hardware processes, allowing them to communicate with the rest of the system easily and systematically.

We are focusing on MOLEN programming paradigm and considering the FPGA as a co-processor rather than having complete hardware threads as ReconOS and BORPH propose. In this regard, our work is more similar to the HybridOS [5] approach in which the granularity of the computation on the FPGA is based on multiple data parallel kernels mapped into accelerators to be accessed by multiple threads of execution in an interleaved and space-multiplexed fashion. The difference between our work and the HybridOS approach is that our work is based on the MOLEN programming paradigm in which an operation, executed by the reconfigurable fabric, is divided into two distinct phases: set and execute. MOLEN defines a minimal ISA extension to support the programming paradigm. Such an extension allows the mapping of an arbitrary function on the reconfigurable hardware with no additional instruction requirements.

StarPU [6] offers a unified task abstraction named "codelet". StarPU takes care to schedule and execute those codelets as efficiently as possible over the entire machine. In order to relieve programmers from the burden of explicit data transfers, a high-level data management library enforces memory coherency over the machine: before a codelet starts, all its data are transparently made available on the computing resource.

We are targeting the tightly coupled processor coprocessor MOLEN paradigm. The MOLEN polymorphic processor [7] consists of a general-purpose processor (GPP) tightly coupled with a reconfigurable coprocessor (RP). The latter can be used to implement arbitrary functions in hardware using custom computing units (CCUs). The programming model for a reconfigurable platform must offer an abstraction of the available resources to the programmer, together with a model of interaction between the components. The MOLEN programming paradigm [8] abstracts the hardware and allows the programmer, the compiler and the run-time system to efficiently use the underlying hardware. A task, executed by the RP, is divided into two distinct phases: set and execute. In the set phase, the RP is configured to perform the required task and in the execute phase the actual execution of the task is performed. This decoupling allows the set phase to be scheduled well ahead of the execute phase, thereby hiding the reconfiguration latency.

Although our work is based on the MOLEN programming paradigm, *set* and *execute* are not microcode in our system. They are APIs provided by the operating system with the same functionality; *set* API means reconfiguring the RP and *execute* API means the actual execution. In fact, we use these two APIs to abstract the concept of task. This way, we can call a task without mentioning the exact implementation. It is the runtime systems responsibility to resolve the called task to a proper implementation.

3. THE RUN-TIME ENVIRONMENT

The presented runtime environment is depicted in figure 1. The Schedulers goal is to end up with certain tasks that can be accelerated on the RP and the remaining parts of the application will be executed on a regular general-purpose processor. Task scheduling can have different conflicting objectives, like improving performance, power consumption or the memory footprint. One scheduling policy is described in section 5.

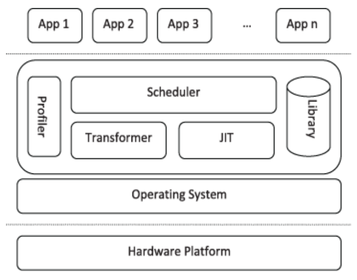

Figure 1. the Run-time Environment

The Profiler continually tracks the application behavior and records statistics about computational intensity and number of referrals. The scheduler can use this information to identify computation intensive tasks and execute them in hardware. Since the runtime profiler has to run in parallel with the actual program execution, it must be very lightweight and low overhead. This requirement differentiates such a runtime profiler from the common profilers such as gprof, etc. More details on the runtime profiler are covered in section 4.

When the scheduler decides to run a task in hardware, the transformer has to replace the software implementation of the task with a call to the hardware implementation. In fact, binary rewriting imposes a considerable overhead to the system and to avoid that overhead we use the MOLEN compiler in which the compiler puts the *set* and the *execute* APIs in the program code. As we mentioned before, these APIs are an abstraction mechanism that separates the task calls from the task implementation. The task binding mechanism is explained in section 6.

The kernel library is a precompiled set of common tasks implemented for a specific architecture. For each task, multiple versions can exist, each with different characteristics. These characteristics are saved as metadata and can contain, among other things, the configuration latency, execution time, memory bandwidth requirements, power consumption and physical location on the reconfigurable fabric. A just-in-time compiler can be used to compile the tasks for which there is no implementation in the library. The compiler converts binary to bit stream. The discussion about JIT compiler is out of the scope of this paper.

4. RUNTIME PROFILING

In this section, we discuss the implementation of our runtime profiler in more details. Based on methods of gathering information, profilers can be categorized into two major groups, Instrumentation-based and Sampling-based profilers.

Instrumentation-based profilers work by inserting instrumentation code into the application to be profiled. The instrumentation code can be injected either at compile time, at link time or at run time. The advantage of such Profilers is that they are very accurate and more portable than other kinds of profilers, but on the downside, the instrumentation phase has a considerable overhead. That's why we perform the instrumentation phase at load time to prevent application speed down during runtime.

Sampling-based profilers periodically take samples of program's Program Counter value. Therefore, the gathered data is a statistical approximation. Although they are not as accurate as Instrumentation-based Profilers, they have much lower overhead, because no extra instruction is inserted in original code. As a result, they are relatively non-intrusive, and allow the profiled program to run almost at its normal speed.

We assume the list of tasks to be profiled is available to us and hence we only need to keep count of those tasks and neglect the others. This assumption is based on the idea of having a library of different function implementations.

4.1 Instrumentation Profiler

The instrumentation profiler must dynamically inject the profiling code to the application binary code. We use a technique inspired by Detours [10] library. Detours is a library which is used to instrument Win32 functions on x86 machines. Detours replaces the prologue of the target function with an un-conditional jump to a detour function, which is provided by the user. Similarly, our code injector replaces the prologue of each function that has to be profiled, with an unconditional jump to the instrumentation code. The only limitation to this technique is that the function prologue size must be equal to or bigger than the unconditional jump instruction size in the target architecture. However since functions of our interests are usually very big, this is not a serious concern.

The purpose of the injected code is to record the function counts. This is a very simple bunch of code in which, the value of a counter for that specific function is incremented. Besides the code also contains the instructions, which were removed from the prologue of the profiled function. In the end, an unconditional jump, jumps back to the remaining part of the profiled function. The instrumentation code is generated offline. Then it is updated with the new addresses at load time and injected into the profiled program.

We have used the *mmap* function to add the extra code to the running program. For this purpose, first, an object file consisting of instrumentation code is created for each profiled process. The object file is a copy of the instrumentation code file, which was generated offline, but with a unique name for each process. This file is then injected into the profiled process through the *mmap* system call. The *mmap* function must be called from within the running program. Therefore, we use the *ptrace* facility in Linux to attach to the running program. After attaching, we stop the program and insert the code that calls the *mmap* function in the program. Before inserting that code, we backup the original data and as well as the registers. The code that executes *mmap* function also has a breakpoint in the end, so that it can stop and hand control back to *ptrace*. After performing those steps, we set the instruction pointer to point to the *mmap* function and let the program execute those instructions until it stops at the breakpoint. When that happens, the original data is placed back in the memory, and registers once again assigned the original values and the original program allowed continuing by detaching *ptrace* from it. At this point, the profiled program contains the instrumentation code and is able to record function counts during its execution. All the addresses within the instrumentation code must be resolved. The *mmap* function returns the starting address of the mapped object file and we use that address to resolve all the addresses in the injected code.

As discussed previously, in our system, we know beforehand the number of functions that can be profiled. We have exploited that restriction to aggressively optimize the instrumentation part of our profiler. This has been made possible by using shared memory. A file of fixed size is shared in the memory, so that different programs can write the function counts to it. That file consists of an array of function counts indexed by function numbers. The instrumentation code for a function, directly increments the count in the shared memory. Since, the increment is done through only a single instruction; it is atomic and does not pose a problem of race condition in cases where different threads are trying to write to the same memory location in the shared memory.

4.2 Sampling Profiler

We are not only interested in the number of times different functions are called, but also approximate time spent per function. It is preferable to transform larger functions than smaller functions. For this purpose, we perform sampling. We have programmed the processor to receive 250 timer interrupts per second when busy. Since we are using a tick-less Linux kernel, in idle conditions the number of interrupts is much lower. A sample contains the Process ID and the instruction pointer value at the time of interrupt. Through that information, we find out the function, which was being executed at the occurrence of the timer interrupt and we get to know, which functions are taking more time to execute. It should be noted that this method does not give very accurate values, because the timer interrupt may hit some functions more often than others.

As the list of the profiled function is known, we can extract the address ranges of each function in each program at load time. By ranges, we mean the starting address and ending address for functions that have to be profiled. The profiler maintains a hash table which is indexed by the process ID. A sample contains the process ID and instruction pointer value at time of interrupt. The Process ID is used as key and the instruction pointer value is checked to be within range of any profiled functions. If the instruction pointer value is found within the range of any profiled function, the sample count for that function is increased. This is done through binary search, since it is the most efficient method for this purpose.

From the function counts and the sample counts for that function, we can approximate the time spent per call for that function. Suppose, the daemon collected 250 samples in a second, and out of those 250, 25 were from a function A. That means that approximately 1/10th of a second were spent overall inside function A. Now, if the function counts for A during the last second was five, then that means, that each call of function A, on average took 1/50th of a second or 20 milliseconds. We keep this information (function counts and time spent per function) in the memory, which is subsequently read by the scheduler to transform the hottest kernels.

4.3 Evaluation

To evaluate the performance of our profiler, we tested it against four different programs. Three of those programs were used to test the performance of instrumentation profiling and one program was used to test performance of sampling profiling. The reason we have more thoroughly tested the instrumentation profiling than the sampling profiling is that it is far more important in the decision-making by the scheduler. The sampling is done just to help the scheduler to identify which functions are too small and therefore not to transform them.

For Instrumentation profiling, we first tested our profiler with a program, which repeatedly calls a function that just adds two numbers. The reason to use this program is to check the performance of the profiler for the worst-case scenarios. The results are shown in Table 1. We can see that even for such a small function, the overhead without sampling is just 6.3%, while with sampling it is 7.2%. Next, we checked profiling overhead with a program that repeatedly calls a function that sums 25 numbers. As it is shown in Table 2, the overhead is less than one percent and is negligible. Finally, we tested it against an industrial scale SAT solver, minisat [11] and for that the overhead is also negligible as shown by Table 3.

Therefore, our profiling system is at par with Dynamo [12], which has overhead of less that 1.5% and better than the profiler presented in [15] which has average overhead of 3%. The reason behind this is that we have the advantage of only profiling a selected number of functions, due to which we have aggressively optimized our profiler. Also from the tables, we can see that sampling adds very little overhead in our profiler, and is better in performance than profilers like [16], which adds 1-3% overhead for sampling. This can be attributed to our limited application, unlike in case of [16] which profiles the entire system (user programs, OS kernel etc...) and not just selected user programs, and also caters other events like cache misses.

Table 1. Profiled Add Performance

Mode	*Profiled (sec)*	*Unprofiled (sec)*	*Overhead*
Without Sampling	30.92	29.09	6.3%
With Sampling	31.18	29.09	7.2%

Table 2. Profiled Sum Performance

Mode	*Profiled (sec)*	*Unprofiled (sec)*	*Overhead*
Without Sampling	31.70	31.76	less than 1%
With Sampling	31.93	31.76	less than 1%

Table 3. Profiled minisat Performance

Mode	*Profiled (sec)*	*Unprofiled (sec)*	*Overhead*
Without Sampling	28.50	28.49	less than 1%
With Sampling	28.23	28.49	less than 1%

For checking the accuracy of Sampling, we used a program in which two functions, which contain exactly the same code, are called different number of times one of which is called 60 times more than the other is called. The sample counts of the function, which is called 60 times more, were 49 times more than the other function. Considering that the time spent in the more frequent function do not require to be exactly 60 times more than the other function, due to features of modern processors, the accuracy is sufficient for our purpose. Lastly, we also measured the time taken to inject code into the instrumented process. We measured the time for two different programs. One program contained four functions, while the other program was minisat, which contains 54 functions. The injection time for both was almost the same and around 60 milliseconds. Therefore, the time was not much dependant on number of functions to be profiled. Most of that time can be attributed to the injection of instrumentation code file and shared memory. The time should be almost the same for any program.

5. TASK SCHEDULING

In our system, task scheduling takes place in two phases. Firstly, at compile-time, the compiler performs static scheduling of the reconfiguration requests assuming a single application execution. The main goal in this stage is to hide the reconfiguration delay by configuring them well in advance before the execution point. Then at run-time, the run-time system performs the actual task scheduling. In this stage, the set and execute instructions are just a hint to the run-time system and they do not impose anything to the runtime system. The run-time system decides based on the dynamic of the system and it is possible to run a kernel in software even though the compiler already scheduled the configuration. The goal of the compiler scheduling is to minimize the impact of the reconfiguration latency over the application performance. The reconfiguration latency is a major drawback in the reconfigurable computers. However, by configuring the tasks well in advance, we can reduce the effect of reconfiguration on the overall application performance. We employed the same approach as presented in [8] to do the static scheduling of the configurations.

At runtime, to overcome the limitation with the size of reconfigurable fabric, a replacement policy is used. This replacement algorithm determines which part of the logic area should be replaced whenever some space is needed. It could be a nice attempt to guess which tasks are likely to be needed soon and if they are already configured, not to replace them with other tasks. Many decision parameters can be used to decide for the replacement such as the frequency of use in the past, the distance from last call, or even a random selection. These parameters are sort of heuristics to keep the best kernel configured. Having information about the future is indeed a real success factor in this decision. In this paper, we use two decision parameters: the distance to the next call and the frequency of the calls in future that can be obtained from a Configuration Call Graph (CCG) generated by the compiler. More detail about CCG can be found in [13]. Furthermore, in our evaluation part, we use a past frequency algorithm in which the required information is obtained from the runtime profiler. This algorithm is similar to the presented algorithm in [9].

At each scheduling point, we define the d_i for task T as the number of task calls between the current execution point and the next call to the task T, in the breadth first traverse of the CCG. We use the minimum value of d_i as the distance to the next call for task T. Similarly, for a specific task T, we define the f_i as the total number of calls to the task T in all the successors of the current execution point. The frequency of the calls for the task T in the whole system is the sum of the all f_i.

One of the main issues in calculating the distance and frequency is the time overhead. To avoid a long traversal of the CCG, a maximum traverse depth is specified to restrict the level of movement in the CCG. To identify the maximum traverse depth, we performed experiments. Figure 2 shows the total execution time of the applications by increasing the traverse depth for the different workloads. More details about our experimental setup can be found in section 5.2. As it is shown in Figure 2, by increasing the traverse depth, the execution time is approaching to a fixed value. At the beginning of each diagram, the execution time varies a lot. However, after around traverse depth of 10, the execution time is getting more flat. Therefore, we set the maximum traverse depth to 10. This means that in all of our traversal algorithms we traverse at most 10 levels of CCG from the current execution point.

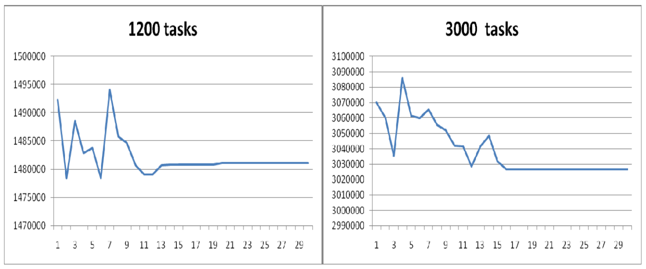

Figure 2. Maximum Traverse Depth

We should mention that in our traversal algorithms for calculation of the distance and frequency, we traverse the node with the highest probability in the alternative execution nodes (OR nodes). Furthermore, in case of coming across a loop, we traverse the loop only one more time (two times in total).

5.1 The algorithm

The run-time scheduling procedure is a two level mechanism. The first phase is a normal scheduling policy for example a normal round robin. At the first level of scheduling, the tasks are being scheduled to run on the general-purpose processor. Apart from that, there are some points that the system needs to decide to use the reconfigurable coprocessor for computation intensive tasks. These points are either the set or execute or the internal events from the run-time profiler.

At each scheduling points, there are a few possibilities. The first choice is to do nothing and continue the execution on the general-purpose processor. The other option is to choose one of the hardware implementations of the kernel from the library, configure it on the FPGA and start hardware execution. Remember that there might be other conflicting tasks already configured on the FPGA and some of them might be in the execution status (busy).

Listing 1. The scheduling procedure at a certain scheduling point deciding about HW/SW execution of task *T*

```
1- Scheduling begins
2- Assume implementation_list is a list of all the
      possible implementations of T;
3- For each $I_i$ in the implementation_list
   If the corresponding physical location is busy
      (running), remove $I_i$ from the list;
4- Assume AlreadyConfigured_list is an empty list;
5- For each $I_i$ in the implementation_list
   If $I_i$ is already configured, add it to the
      AlreadyConfigured_list;
6- If AlreadyConfigured_list is not empty
      6-1- Find $I^*$ with the minimum execution time in
                AlreadyConfigured_list;
      6-2- If scheduling status is execute, execute $I^*$
      6-3- GOTO 9;
7- If implementation_list is not empty
      7-1- Find $I^*$ with the minimum (configuration_latency
                +execution_time)in implementation_list;
      7-2- Remove $I^*$ from the implementation_list;
      7-3- Assume ToBeEvicted_list is an empty list;
      7-4- Add all the kernels which are already
           configured on the FPGA and have overlaps with
           $I^*$ physical location to the ToBeEvicted_list;
      7-5- For all $I_j$ in the ToBeEvicted_list, if there is
           a $I'$ with a distance to next call smaller than
           the distance to the next call of $I^*$ go to 7;
      7-6- Configure $I^*$ on to the FPGA;
      7-7- If scheduling status is execute, execute $I^*$
      7-8- GOTO 9 ;
   8- Continue with software execution
   9- Scheduling ends
```

Listing 1 presents the scheduling procedure in more details. In Step 3, the scheduler must ensure that in the physical location of each implementation on the FPGA, there is no other busy task configured. In Steps 5 and 6, the scheduler checks if any of implementation is already configured and is in ready status. This means there is no need for reconfiguration and the hardware execution can start right away. In step 7, the scheduler is trying to find a replacement candidate. Considering steps 6-1 and 7-1, it is clear that we are optimizing the execution time. As we mentioned before, other optimization can be also applied such as selecting the task with smaller size or less energy consumption and so on.

The replacement decision is being taken in step 7-5. In the listing 1, we used the distance to the next call as our decision parameter. Steps 6-2 and 7-7 show the difference between *set* and *execute* instructions. When the scheduling procedure is called because of a *set* instruction, it means the actual execution will be later. Therefore, in this case, all the steps are followed except the execution steps 6-2 and 7-7. For the *execute* instruction, all the steps is followed again. If the kernel has been already set by the previous instruction and it is in ready statues, the execution can start right away without any configuration delay. However, in a multi application scenario, other applications might have influence on the configuration. For example, applications might share the same kernel and the kernel which is set by this application can be used by another one in between the set and execute instructions. Therefore, following the scheduling procedure again at the execute instruction is necessary. From the scheduling algorithm, it is clear that the scheduling is non-preemptive and there is no task migration from CPU to FPGA. As soon as a specific task starts on the CPU, it cannot be migrated to the FPGA anymore in the same run.

5.2 Performance Evaluation

In this section, we compare three algorithms, which are based on the distance to the next call, frequency of calls in the future and frequency of calls in the past. The frequency of calls in the past is calculated based on the information provided by the runtime profiler. We use our discrete event simulator which is an extension of the CPUSS CPU scheduling framework [14] in order to evaluate the algorithms. The simulation workloads are generated after carefully analyzing various hardware kernels. In our workloads, the hardware implementation of a task is in average 5 times faster than the software implementation. Each application consists of 50 tasks including redundant tasks (an application can have between 5 to 20 unique tasks). The tasks can be shared between applications. We assume each task has at least one hardware implementation. In average, each task has three different versions of hardware implementations. It should be mentioned here that the data dependencies between the tasks within one application should be managed inside the application and the run-time system cannot do anything about it. In fact, the application sends the requests for task execution and the request should not be issued unless all the dependencies are met.

Table 4 shows the obtained results for each algorithm which certify that the distance to the next call is a better parameter to decide on. Additionally, the results show that the frequency of calls does not perform as well as the distance to the next call. The reason is that although a kernel might be used several times in the future, it does not mean that those uses are in the near future. For example, a task might be accessed 10 times during 10 seconds from which one access is in the first one second and the rest are in the last 2 seconds. Using highest frequency algorithm suggests keeping this task on the FPGA, however seven seconds between the calls might be enough time to remove this task from the FPGA and bring it back later on. Despite this fact, we believe that the frequency is a good decision parameter especially for the workloads with periodic execution behavior. The reason for better performance of future frequency over the past frequency is also obvious. Looking at the past frequency is a kind of heuristic to predict the future and it is not always accurate. Using the CCG in the future frequency algorithm, we have a better heuristic with more accuracy.

Considering overall application performance compared to the software only execution, we obtained speedup of 2.29. As mentioned before the hardware implementations of the task are, on average, 5 times faster. The past frequency algorithm also obtained speedup of 1.66. In the ideal case with our assumption, the speedup should be around five if all the tasks run in hardware.

Table 4. The percentage from the software execution time

Number of Applications (Number of tasks)	Distance to next call	Frequency in future	Past Frequency	Software Only
3 (150)	32%	33%	37%	100%
6 (300)	36%	37%	44%	100%
12 (600)	52%	55%	61%	100%
60 (3000)	44%	51%	60%	100%

6. DYNAMIC BINDING

As we pointed earlier, the actual binding of the function calls to the implementations happens at runtime. For each task's implementation in the library, there is a software wrapper function, which performs the configuration and execution of the operation. In fact, the *set* and *execute* APIs are a call to this wrapper. The configuration and execution can be done at the same time or they can be done separately based on the provided arguments. This is because of the possible distinct *set* and *execute* in the MOLEN programming paradigm. The software wrapper is kept in the form of a Dynamic Shared Object (DSO). The application developer can also provide his own DSO along with the required metadata. Given the name of a DSO by the scheduler, the system dynamically loads the object file into the address space of the program using the Linux *dlopen* facility and returns a handle to it for future operations. To find the address of the wrapper function in the DSO, we use Linux *dlsym* function. This function takes the name of the function and returns a pointer containing the resolved address of the function. Based on the provided arguments, the wrapper function can either configure the computation in the FPGA, execute it or both.

6.1 Evaluation

The execution time overhead imposed by dynamic linking (DSO loading) occurs on two places: at run and load-time. At runtime, each reference to an externally defined symbol must be indirected through the Global Object Table (GOT). The GOT contains the absolute addresses of all the static data referenced in the program. At load-time, the running program must copy the loaded code and then link it to the program. In most cases, the only run-time overhead of dynamic code is the need to access imported symbols through the GOT. Each access requires only one additional instruction. The load time overhead is the time spent to load the object file. For a null function call in our system, the load time is about 0.75 milliseconds. For a typical wrapper function, the load time increases to about 2 milliseconds. We should mention that the increase in the input parameters' size might increase the size of the wrapper function since each parameter needs a separate instruction to be transferred to the MOLEN XREGs.

7. CONCLUSION

In this paper, we proposed our runtime environment which is responsible to assign tasks to the reconfigurable accelerators. It consists of a scheduler, a runtime profiler and a transformer. These components together enable the system to recognize the computation intensive tasks, allocate the proper implementation on the FPGA and execute it. The profiler, records application behaviors at runtime and stores statistics about the application execution. We showed the overhead imposed by the profiler is less than one percent of the application execution time. We also presented the runtime system scheduler and as a case study we presented a few scheduling algorithms performance of which is shown by experiments. Moreover, we presented our runtime binding mechanism, which is used to bind a task call to a task implementation with a negligible overhead.

8. ACKNOWLEDGMENTS

This research is partially supported by hArtes project EU-IST-035143, Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230) and FP7 Reflect (grant 248976).

9. REFERENCES

[1] Huang, S.S., et al., Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary, *European Conference on Object-Oriented Programming (ECOOP)*. 2008.

[2] Olukotun, K. Towards Pervasive Parallelism. 2010; Available from: http://ppl.stanford.edu/wiki/index.php/Projects.

[3] Lübbers, E. and M. Platzner, ReconOS: An Operating System for Dynamically Reconfigurable Hardware, *Dynamically Reconfigurable Systems*. 2010, Springer.

[4] So, H.K.H., R. Brodersen, A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems*, 2008. 7(2).

[5] Kelm, J. H. and Lumetta, S.S., HybridOS: Runtime Support for Reconfigurable Accelerators, *In Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2008.

[6] Augonnet, C., Thibault, S., Namyst, R., Wacrenier P.A., StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Proceedings of the 15th International Euro-Par Conference on Parallel Processing,* 2009.

[7] Vassiliadis, S., et al., The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 2004. 53(11): p. 1363-1375

[8] Panainte, E.M., K. Bertels, and S. Vassiliadis, The Molen compiler for reconfigurable processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007. 6(1).

[9] Fu, W. and K. Compton. An execution environment for reconfigurable computing. *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 2005.

[10] Hunt, G. and D. Brubacher, Detours: Binary interception of win32 functions, *3rd USENIX Windows NT Symposium*. 1998.

[11] minisat. Available from: http://minisat.se/.

[12] Bala, V., E. Duesterwald, and S. Banerjia, Dynamo: a transparent dynamic optimization system, *Conference on Programming Language Design and Implementation*. 2000.

[13] Sabeghi, M. and K. Bertels, Toward a Runtime System for Reconfigurable Computers: A Virtualization Approach, *Design, automation and test in Europe*. 2009.

[14] Barnett, G. CPU Scheduling Simulator. cpuss.codeplex.com, 2009.

[15] Arnold, M. and B.G. Ryder, A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 2001. 36(5)

[16] Anderson, J.M., et al., Continuous profiling: where have all the cycles gone?, *ACM Operating Systems Review*, 1997. 31(5).