Efficient Runtime Support for Embedded MPSoCs

Dimitris Theodoropoulos
dtheodor@ics.forth.gr

Polyvios Pratikakis
polyvios@ics.forth.gr

Dionisios Pnevmatikatos
pnevmati@ics.forth.gr

Computer Architecture and VLSI Systems Laboratory
Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH)
100 Plastira Avenue, Vassilika Vouton
GR-70013 Heraklion - Crete, GREECE

***Abstract*—Recently, many software runtime systems have been proposed that allow developers to efficiently map applications to contemporary consumer electronic devices and high-performance academic processing platforms. Most of these runtime systems employ advanced scheduling techniques for automatic task assignment to all available processing elements. However, they focus on a particular environment and architecture, and it is not easy to port them to reconfigurable embedded MPSoCs. As a consequence, in the embedded community, researchers implement hardwired application-specific task schedulers, which can not be used by other embedded MPSoCs. To address this problem, in this paper we propose a lightweight runtime software framework for reconfigurable shared-memory MPSoCs, that integrate a master embedded processor connected to slave cores. Similarly to many of the aforementioned advanced runtime systems, we adopt a task-based programming model that uses simple, pragma-based annotations of the application software, in order to dynamically resolve task dependencies. Our runtime system supports heterogeneity in the hardware resources, and is also low-overhead to account for possible limitations in their processing capabilities and available on-chip memory. To evaluate our proposal, we have prototyped an MPSoC with seven slaves to a Xilinx ML605 FPGA board. We run three micro-benchmarks that achieve a performance speedup of 3.8x, 7x and 5.8x, and energy consumption of 27%, 14% and 18% respectively, compared to a single-core baseline system with no runtime support.**

***Keywords*-runtime support, embedded MPSoCs, FPGAs**

I. Introduction

Recent consumer electronic devices integrate an increasing number of processing cores, accelerators, etc. Many general purpose, high-performance platforms tightly couple contemporary CPUs with GPUs [1] and/or FPGAs [2]. These devices are frequently used for scientific applications that require very powerful processing. A significant challenge though for the developers, is to efficiently map applications to all available processing cores.

To harness the processing power of such systems, computer science research communities have proposed software frameworks, frequently called runtime support, that are responsible for automatic application tasks assignment to all available processing elements. In most cases, developers are only required to annotate with specific key-words certain application software segments, which help the runtime to efficiently map them to the underlying hardware. An important benefit is that development times are significantly reduced at the expense of a processing overhead introduced by the runtime, due to the tasks scheduling.

Most of the proposed runtime systems in the literature employ heavyweight features to support the aforementioned, powerful platforms. Examples are various code-annotation clauses for task synchronization and scheduling [3], support for different processing platforms [4], and internal advanced scheduling algorithms, such as work-stealing, tasks input arguments locality analysis and tasks multi-issuing mechanisms [5].

However, there are many academic and industrial proposals that *do not utilize such advanced and powerful hardware platforms*. In contrast, researchers develop simpler application-specific Multi-Processor System-on-Chip (MPSoCs) within a single or few reconfigurable chips with embedded soft- and hard-core processors and/or custom acceleration modules as building blocks. Such embedded systems oftentimes are heterogeneous, but lack considerably of processing capabilities and available on- and off-chip memory compared to the aforementioned high-performance platforms. Most general-purpose runtime systems are not a good match for such systems, and consequently, in most cases, the developers implement application-specific task schedulers, which cannot be applied to other application domains.

Research on literature shows that other already proposed runtime systems for embedded MPSoCs, require real-time operating systems (e.g. uCLinux), and interrupt support from all worker processors [14], [15]. Such features are mainly utilized for real-time task scheduling to available worker cores, to improve the system performance, avoid thermal hot spots, and provide fault tolerance when transient or permanent errors occur. However, they can be prohibitive for many embedded MPSoCs, since they do not integrate

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the DeSyRe Project (www.desyre.eu), grant agreement number 287611.

enough resources to host a real-time OS or support interrupts. Consequently, these runtime systems cannot be easily ported to the aforementioned platforms, which is another reason why developers implement application-specific task schedulers that cannot be applied to other application domains.

To address these issues, we propose a lightweight runtime framework that targets reconfigurable, heterogeneous embedded MPSoCs. Our work is an effort to provide general purpose runtime support for reconfigurable MPSoCs that consist of embedded processors and/or custom hardware modules. Certain advanced runtime features that would introduce significant processing overhead are omitted, thus our proposal can be utilized by such constrained MPSoCs. More specifically, the paper contributions are the following:

- We present a runtime framework for reconfigurable heterogeneous embedded MPSoCs, which allows rapid application development and efficient tasks allocation to all available processing elements. Our work adopts a task-based programming model that requires the developer to simply pragma-annotate the application code, in order to dynamically resolve task dependencies.
- We support our high-level programming interface with a C-to-C Java compiler that converts the pragma-annotated code to source files, which include all required runtime API calls. Moreover, our compiler resolves all Write-After-Write (WAW) and Write-After-Read (WAR) task dependencies by renaming their outputs.
- We have implemented an actual FPGA-based MPSoC that consists of one master and seven slave processors, and mapped our runtime software.
- To evaluate our proposal, we run three micro-benchmarks and measured the MPSoC execution time and energy consumption. Based on task granularity, results suggest that our runtime system may introduce minimal processing overheads, while reduce energy consumption compared to a baseline system with no runtime support.

The paper structure is as follows: Section II provides references to related work on runtime support for reconfigurable MPSoCs and applications that use such systems. In Section III, we describe our proposed runtime system, while in Section IV we report our experimental results. Finally, in Section V we conclude the paper and also mention our future work.

II. Related Work

Runtime frameworks: There are many task-parallel runtime systems proposed that target high-performance platforms [5], [4], [3], [6], [7], [25]. An interesting survey on state-of-the-art runtime management can be found in [8]. These systems utilize advanced yet complicated scheduling techniques, such as work-stealing and tasks multi-issuing mechanisms. As they all rely on complex libraries and sophisticated OS features, they cannot be directly used on reconfigurable MPSoCs that are explicitly composed by soft- and/or hard-core processors without any real-time OS support and a large porting effort.

To solve this, Patel et al. [9] present a framework for executing parallel applications on FPGA-based multiprocessors that employ multiple chips. Unlike our work that considers shared-memory MPSoCs, their framework uses a distributed-memory model, where each computing engine is assigned its own local memory. Data among tasks are exchanged using an MPI-compliant [10] implementation. Tumeo et al. present a system similar to ours [11]. The authors describe a shared-memory MPSoC, which employs Microblaze soft-core processors, a multiprocessor interrupt controller (MIC) and a synchronization engine.

Other work considers the employment of a real-time OS [12], [13]. A problem though is that many applications do not need certain advanced features that real-time OSs offer, like network and file system support. In addition, a significant amount of the FPGA computational and memory resources may be utilized by the OS itself, thus leaving less space and resources for the user application. For example, in [14], the authors propose a task management software infrastructure tailored to MPSoC with distributed operating systems. Similarly to our work, the paper considers a master / worker approach that employs a shared-memory, however the entire system runs a distributed version of the uCLinux OS. In [15], the authors present a runtime system that also considers a shared-memory architecture, where each processing core runs its own uCLinux instance.

The MADNESS project [16] also focuses on task dynamic remapping and employs a runtime system that can execute applications described using the Polyhedral Process Network (PPN) [17]. Finally, in [18] the authors present a framework for automatic generation of runtime managers that target reconfigurable MPSoCs. Similarly to our work, they consider a task-based programming model that uses OpenMP clauses.

Application-specific approaches: There is a considerable amount of work that employs reconfigurable MPSoCs to exploit an application's potential of parallel process [19]. Zhou and Freear [20] describe an MPSoC for fast real-time simulation of complex and nonlinear wheel-rail contact mechanics, which uses six NiosII embedded soft-core processors. Dykes et al. [21] present an MPSoC for a laser-based transparency meter, implemented using three Microblaze soft-core processors that execute custom software kernels.

Karanam et al. [22] target the Smith-Watterman algorithm using 14 Microblazes for data processing and one Microblaze for the overall system control. Similarly, Mplemenos and Papaefstathiou [23] target the BLAST algorithm using an MPSoC consisting of 80 Microblaze cores. Theodoropoulos et al. [24] present an FPGA-based MPSoC that employs two PowerPCs in combination with custom accelerators

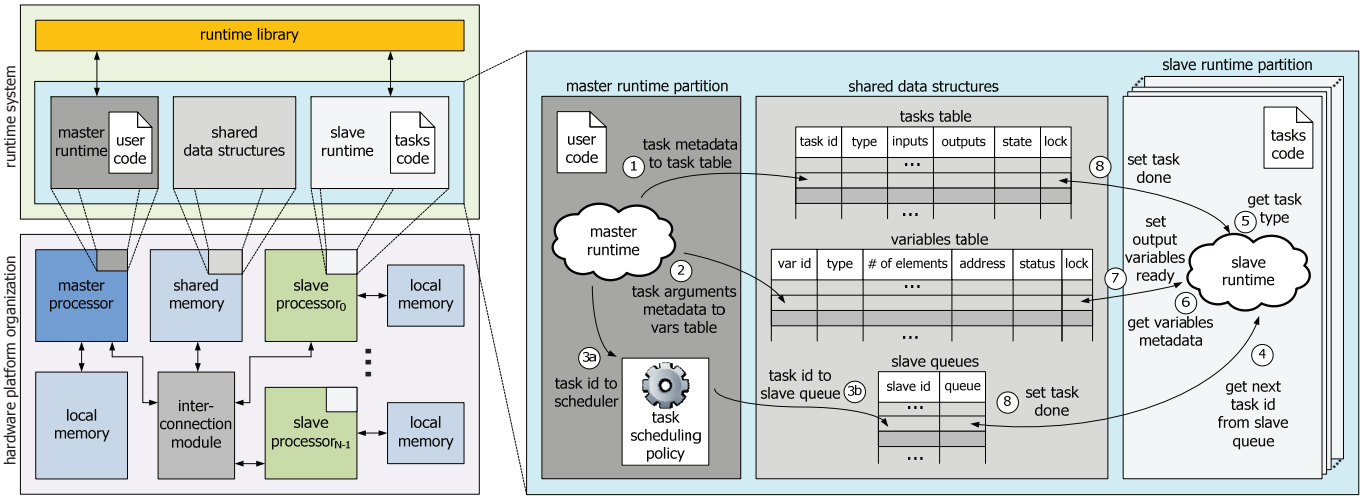

Figure 1. The runtime structure and its mapping on a shared-memory MPSoC architecture. Rounded numbers indicate the required runtime steps, in order to assign a task to a slave core; steps 1-3b are executed by the master runtime to generate a new task and add it to the shared resources, steps 4-6 are executed by the slave runtime to start a specific task, and steps 7-8 update the shared resources when a task is complete.

for audio acquisition and rendering. Ravindran et al. [26] present an IPv4 Packet forwarding application implemented to an FPGA-based MPSoC that employs two PowerPCs and 14 Microblaze processors for payload and header packet processing.

Overall, many research groups employ FPGAs to rapidly implement MPSoCs for different application domains. However, a major development bottleneck is the software/hardware co-design, since it requires careful synchronization between them. Our work contributes to resolving this problem, since it combines a high-level software interface with a lightweight runtime support that efficiently maps all application tasks to the MPSoC processing elements.

III. The Runtime System

Runtime structure: We aim to provide a high-level software environment that allows the developers to easily parallelize and map their applications to reconfigurable MPSoCs. As shown in the bottom-left part of Figure 1, we target MPSoCs that are based on a shared memory model. Moreover, we consider a structure with one master and several slave processors that communicate via an interconnection module, like buses or NoCs. Note that our runtime system does not require slave processors to integrate interrupt support, a feature which considerably relaxes their specifications and complexity. Consequently, slave processors may be open-source or vendor-specific, soft- or hard-core, or any type of hardware modules that have been assigned a globally-mapped address space.

The master and slave processors have also their local data memories. In the top-left part of Figure 1, we show the overall runtime structure. All internal module interactions described below, are illustrated in its right part. The runtime system consists of four main modules, namely *runtime library*, *shared data structures*, *master runtime* and *slave runtime*.

The *runtime library* consists of a set of runtime functions, used during the application execution for assigning tasks to slaves and checking dependencies. It also includes functions that help in system debugging. The *runtime library* is used both by the *master runtime* and *slave runtime* partitions.

The *shared data structures* store runtime data that need to be shared both by the master and all slave processors. As illustrated in Figure 1, during application execution the runtime generates a task-table that contains all tasks and their metadata, such as a unique id, type (each annotated task defines a certain type of task instances), inputs/outputs variable ids, current state (issued, executing, done) and its lock. The latter is used to make sure that at any time only a single slave is accessing a task's metadata. In addition, the runtime creates a variables table, where in each of its entries are stored metadata regarding the application variables. These are a unique variable id, type (e.g. short, int, float, etc.), number of array elements, starting memory address, current status (ready or not to be consumed), and a lock for forbidding simultaneous access of the variable data by more than one processors. Finally, the runtime also generates a task queue for each slave processor, used to store the task ids that are assigned to the corresponding slave processor for execution.

The *master runtime* is the primary runtime partition executed on the master processor. It includes the user application code along with the task scheduling API calls to the runtime library, while it also initializes all shared data structures. As soon as a task call is identified by the runtime, the task metadata are stored in the tasks table (step 1). Its input/output variables metadata are stored in the variables table (step 2). Then, the runtime sends the

```
#pragma worker_task l1_norm_task: in_types:float out_types:float
{
  void l1_norm_task(float *x, float *x_l1_norm) {
    int i, k;

    for(k = 0; k < ARRAY_LINES_PER_SLAVE; k++) {
      x_l1_norm[k] = 0.0f;
      for(i = 0; i < N; i++) {
        x_l1_norm[k] = x_l1_norm[k] + fabs(x[i + k * N]);
      }
    }
  }
}
```

Figure 2. Task annotation with input and output types.

task id to the task scheduler (step 3a) and, based on the selected task scheduling policy, the latter assigns it to the slave processors by writing its id to the corresponding task queue (step 3b). This procedure is repeated for every task call, and it may be suspended either if the user has inserted a specific *sync_barrier* instruction in the application code, which waits for all tasks to be done, or the shared data structures resources are full.

The *slave runtime* is the runtime partition that runs on each slave processor. It also uses a certain function set from the runtime library to access its host slave processor task queue and fetch the next task id that needs to be executed. As soon as a valid task id is found in the slave processor task queue (step 4), the *slave runtime* accesses the tasks table to find its type (step 5) and the variables table to retrieve the task input/ouput variables metadata (step 6). Moreover, the *slave runtime* allocates the required space on the slave processor local memory and copies the task input data. Once data transfer is complete, based on the task type, the slave processor invokes the corresponding task software code and process all data. As soon as data processing is complete, the *slave runtime* copies back the results to the shared memory, and sets all task output variables as ready to the variables table (step 7), and the task status to the tasks table and slave queue as done (step 8). Then the *slave runtime* resumes from step 4, in order to process the next task id found in the slave processor tasks queue.

Note that the entire runtime software is written in strict ANSI C. Consequently, it can be ported to any shared-memory reconfigurable MPSoC with minor modifications, and thus be used by any task-partitioned application.

Software framework: Similarly to widely-used advanced runtime systems [6], [3], [4], our work adopts a task-based programming model and requires that developers partition the application code into distinct tasks with certain inputs and outputs. In order to resolve the dependency among these tasks, they also need to introduce a few pragma-based annotations to the original C code. These annotations are done in the tasks' function declaration codes and the main body. Table I summarizes all supported software scheduling clauses of our runtime.

For example, Figure 2 shows how to annotate a task's function declaration. On top of the original code, developers

```
int main() {
  int j;
  float *x, *x_l1_norm;
#pragma system inputs {x}
  x = (float*) malloc(sizeof(float) * N * N);
  x_l1_norm = (float*) malloc(sizeof(float)*N);

  for(j = 0; j < N; j = j + ARRAY_LINES_PER_SLAVE) {
#pragma for-loop(j) task_call l1_norm_task
in_vars{&x[j*N]:BUFFER_SIZE}
out_vars{&x_l1_norm[j]:ARRAY_LINES_PER_SLAVE}
    l1_norm_task(&x[j*N], &x_l1_norm[j*N]);
  }
  free(x);
  free(x_l1_norm);
  return 0;
}
```

Figure 3. Code annotation within the main user application.

insert a single line that contains the *#pragma worker_task*, followed by the function name and the type of all input/output variables. In Figure 2, the task name is l1_norm_task, which has one input variable of type float and one output, also of type float.

Within the main body of the program, developers need to annotate the application input variables, and also describe the input/output variable sizes during the main task call. Figure 3 shows an example of how this can be done. To designate the application input variables that are considered as ready for consumption once the application execution starts, developers introduce a *#pragma system_inputs* code annotation, which includes in brackets all inputs names. In this example, the application input is array x. Also, just before the main task call, a *#pragma for-loop(<loop iterator>) task_call* code annotation is inserted, followed by the task name. Next to the task name, developers need to add the address of the first element of all input/output variables and their sizes, separated by a semicolon sign.

This data is needed by the runtime to resolve which tasks can be directly executed once the program starts, and also identify the task dependencies during the program execution. The *for-loop(<loop iterator>)* is used by the runtime to distribute task instances evenly to each slave starting from the one with the lowest id value. Once the iterator value reaches the maximum number of available slaves, the runtime will again assign another task instance to the first slave and then continue to the next ones.

When the task call is not within a for-loop, the developer simply skips the *for-loop(<loop iterator>)* part. In this case, the current runtime policy is to assign a task instance to the slave with the lowest id and then continue to the next ones. Part of our future work is to include more task assignment policies and evaluate them with more experiments.

We have also developed a software tool-chain that allows programmers to map a pragma-annotated code on a reconfigurable MPSoC. The annotated code is source-to-source compiled from our Java compiler, in order to replace the pragmas with the actual runtime API function calls. Its output is two source files, one for the master processor and

Table I
SUPPORTED SOFTWARE CLAUSES.

Software scheduling clauses	Functionality
#pragma system_inputs{<input vars>}	Definition of application input variables
#pragma worker_task <task name> in_types:<in var types> out_types:<out var types>	Task annotation and arguments type declaration
#pragma task_call <task name> in_vars{<in vars address:size>} out_vars{<out vars address:size>}	Task call with specific input/output variable addresses and sizes
#pragma for-loop (<loop iterator>) task_call <task name> in_vars{<in vars address:size>} out_vars{<out vars address:size>}	Task call within a for-loop with specific input/output variable addresses and sizes
sync barrier()	Suspends task scheduling until all currently executing tasks are done

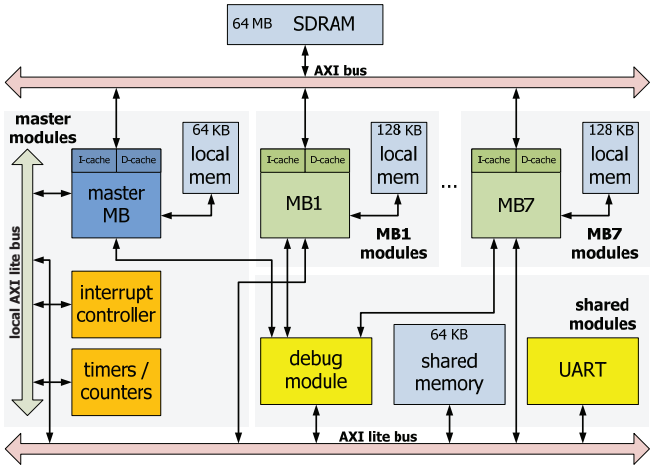

Figure 4. Experimental system with seven slave MBs.

one common file for each slave. Developers may select a predefined task assignment policy from a configuration file, which, as we mentioned above, currently supports only one. Optionally, they may import their custom task scheduling policy if needed. It should be noted that our Java compiler resolves any WAR and WAW dependencies by renaming all task outputs. In addition, developers may select the number of active slaves, in cases where the application workload does not require all slave processors to be active, thus leading to less energy and power consumption. The final step is to apply these files as sources for the master and slave processors using specific vendor tools, and map the final system to the chip.

IV. EXPERIMENTAL RESULTS

Hardware platform: To evaluate our proposed runtime software, we used the Xilinx EDK 13.3 suite to implement a multi-core system. As illustrated in Figure 4, in our current implementation we used a Microblaze soft-core processor as master processor (MMB) to host the *master runtime* paritition. MMB has 8 KBytes of instruction and data cache respectively, while a local 64-Kbyte non-cached memory is also available. In our current *master runtime* version we use the xilkernel as the MMB host kernel, which requires two timers and an interrupt controller to be connected, hence a local AXI-lite bus was employed.

We also connected *seven* Microblazes (SMB) as slave processors, where each one of them also has its 8 KBytes instruction and data cache, and a local 128-KByte non-cached memory. In order to reduce bus congestion, we used two separated AXI buses to interconnect the MMB with all SMBs; an AXI-lite bus where the MMB, all SMBs and an on-chip 64-KByte shared memory are connected, and a high performance AXI bus for the MMB and all SMBs to access the shared off-chip SDRAM. To enhance the system performance, we place all shared data structures in the on-chip shared memory, consequently the *master runtime* and *slave runtime* partitions never go off-chip whenever an update to their data is required. The entire system was implemented and mapped onto an ML605 development board with a Virtex6 FPGA, which occupied 9183 slices (24%), 304 RAMB36E1s (73%) and 24 DSP48E1s (3%).

Runtime evaluation: To evaluate the correctness and efficiency of our proposed runtime system, we ran three micro-benchmarks, a set of cascaded matrix multiplications, an L1-norm matrix computation and a Discrete Fourier Transform (DFT) implementation. Figure 5 shows our custom micro-benchmark with cascaded matrix multiplications. There are four initial matrices, namely A, B, C and D. When the micro-benchmark begins, a series of matrix multiplication tasks are performed, and generate all matrix combinations, as shown in the figure. As soon as all multiplications are done, each pair is multiplied and produces a new matrix, leading eventually to a single output matrix.

An important observation from Figure 5 is that a serial benchmark execution would require 24 multiplications calculated in series. However, due to the fact that our 7-slave MPSoC does not have enough slaves to execute concurrently all 12 multiplications of step 1, an additional step is required. Consequently, the total steps are 6, thus the theoretical micro-benchmark speedup is 24/6 = 4x. The importance of this micro-benchmark is that it includes sequential and concurrent parts (shown within the gray rectangles). Consequently, we can verify the parallel task process extrapolation capabilities of our runtime system, while at the same time all Read-After-Write (RAW) dependencies among tasks are obeyed correctly.

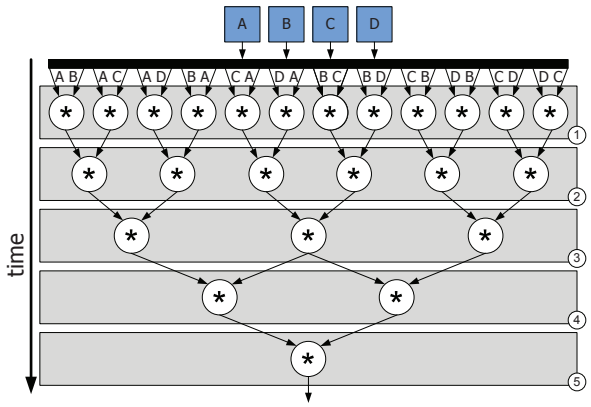

Figure 5. Cascaded matrix multiplications micro-benchmark; the numbers in circles indicate each processing step.

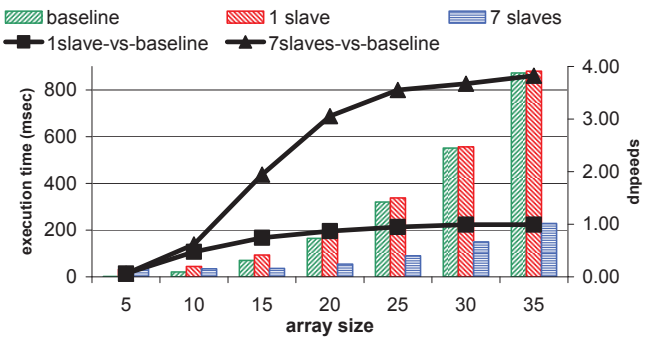

Figure 6. Cascaded matrix multiplications micro-benchmark execution time and speedup against baseline.

The L1-norm calculation of a matrix is the second micro-benchmark that we used; for each matrix row, all its elements are accumulated and stored to the corresponding position of a single-column vector. In order to explore our MPSoC maximum parallel processing capabilities, during our experiments we generated a *constant* number of 7 tasks for all input matrix sizes that were tested. This means that as the matrix sizes increase, the internal task workload becomes larger. Consequently, the theoretical speedup for this micro-benchmark is 7x.

The DFT is our third micro-benchmark; in contrast to the previous two ones, where the number of generated tasks is constant under all inputs (only the task workload changes), in this case varies, and is equal to the number of DFT N-points. Each DFT task computes a complex output sample, and in total N samples are calculated. During our experiments we performed tests with N = 20, 40, 60, 80, 100, 120. To find all required steps from our 7-slave MPSoC to calculate an N-point DFT, we divide N by the # of slaves and round up the result: # of steps = $\lceil \frac{N}{\#ofslaves} \rceil$. Thus, to calculate the theoretical speedup, we divide the N-points by the # of steps, which for all *tested* N-points is 6.67x.

All of our experiments were executed to an actual hardware implementation of the system shown in Figure 4, mapped onto an ML605 FPGA board. To verify the *correctness* of our runtime task scheduling algorithms, we compared our experimental results against the micro-benchmark software implementations executed to an x86 desktop PC. However, it would not make sense to compare the performance of our hardware system against such platform, since they differ considerably in terms of architecture and technology. In contrast, with our experiments *we aim to explore the performance overhead of our runtime* due to its task scheduling operations, compared to an identical system without runtime support. For this reason, all of our experiments were also executed on a baseline hardware system, which integrated a Microblaze under an identical SMB configuration, but without runtime support, and mapped onto our ML605 FPGA board. The baseline performance was compared against two MPSoC configurations, a 1-slave and a 7-slave. The 1-slave utilizes the MMB and only a single SMB, while the the 7-slave employs the MMB and all seven SMBs.

Micro-benchmark 1: On the left y-axis of Figure 6 we show the cascaded matrix multiplications micro-benchmark execution times of the baseline, 1-slave, and 7-slave MPSoCs. On the right y-axis, we provide the achieved speedup of the 1-slave and 7-slave MPSoCs against the baseline. On the x-axis, the array sizes intentionally range from small-sized (5x5) to larger ones up to 1225 elements (35x35), to explore how the overall performance is affected under varied-sized tasks. As we can see from the figure, with a 5x5 matrices size, the 7-slave MPSoC is 94% slower compared to the baseline due to the tasks scheduling overhead.

However, the 7-slave MPSoC starts performing better than the baseline when the array sizes become 15x15 or larger. As we can observe from Figure 6, the 7-slave MPSoC scales to almost 3.8x faster than the baseline when the matrix sizes become 35x35. This means that the task execution time has become considerably larger than the scheduling overhead, thus leading to a performance gain against the baseline, which is close to the theoretical 4x. At the same figure, we can see that under a 20x20 matrix size configuration, the 1-slave MPSoC is 13% slower compared to the baseline, again because of the tasks scheduling overhead. However, as the sizes increase, its performance reaches asymptotically the baseline one, which means that the task scheduling overhead is hidden by the larger tasks execution time due to the higher number of computations per task.

Micro-benchmark 2: Figure 7 shows the performance of the L1-norm matrix computation. On the left y-axis again we provide in a logarithmic scale all execution times, and on the right y-axis our MPSoCs speedup against the baseline. On the x-axis we show all array sizes we used for our experiments, which again intentionally range from small ones (7x7) to larger ones up to 448x448. As we can see from the figure, up to a 14x14 matrix size, the 7-slave MPSoC is slower compared to the baseline, when the size increases to

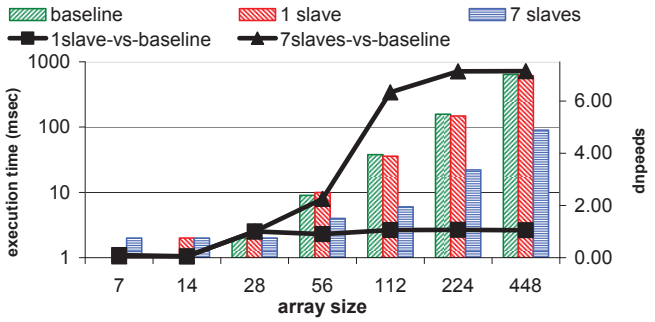

Figure 7. L1-norm micro-benchmark execution time and speedup against baseline.

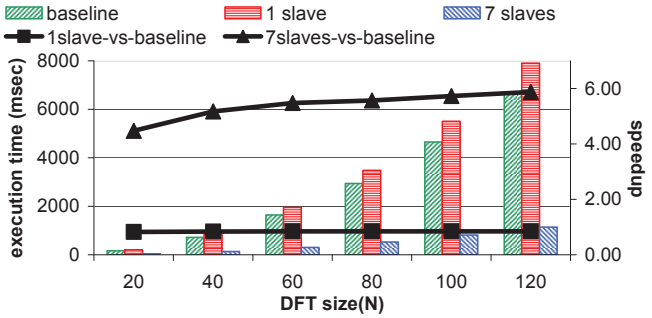

Figure 8. DFT micro-benchmark execution time and speedup against baseline.

28x28, both systems perform almost equally, and for larger sizes, the 7-slave achieves a better performance than the baseline.

The L1-norm micro-benchmark requires significantly less task scheduling overhead compared to the cascaded matrix multiplication one, thus under a time-consuming task partitioning, the 7-slave can reach its theoretical performance. The same applies also for the 1-slave MPSoC, which due to the little runtime overhead compared to the tasks execution time, can also reach its theoretical performance.

Micro-benchmark 3: Figure 8 shows the DFT performance when executed at the 7-slave MPSoC and the baseline, when tested N DFT-points range from 20 to 120. As we can observe, even for a 20-point DFT the 7-slave MPSoC performs 4.5x better than the baseline. Moreover, under a 120-point DFT, the speedup increases to almost 5.8x, which is 13% lower from its theoretical 6.67x. At the same figure we can also see that the 1-slave MPSoC saturates to an 0.85x speedup compared to the baseline, again due to the tasks scheduling overhead.

Energy and power consumption: Performance enhancement is definitely not the only important evaluation parameter of an MPSoC. The latter are found on many portable devices, such as smartphones and tablets, thus energy and power consumption needs to be minimized as much as possible, in order to increase the device autonomy. For this reason, we conducted additional experiments to measure the energy consumption when executing both micro-benchmarks on the 7-slave and the baseline systems.

According to the Xilinx XPower, the 7-slave MPSoC and the baseline require 4.4 Watts and 4.2 Watts of power respectively. Using the micro-benchmarks execution times, we calculated the chip energy consumption with the formula $E = P \cdot t$, where E is the consumed energy, P is the applied power to the chip, and t is micro-benchmark execution time.

Table II shows the % energy consumption of the 7-slave MPSoC compared to baseline one under the largest task size from the experiment setups discussed above. From our experiments we observed that, when the task data sets are small, the 7-slave MPSoC introduces a significant amount of energy consumption due to the runtime overhead,

Table II
ESTIMATED % ENERGY CONSUMPTION FOR ALL MICRO-BENCHMARKS COMPARED TO THE BASELINE UNDER THE LARGEST TASK WORKLOAD.

Micro-benchmark	Task workload	Energy
Cascaded multiplications	35x35 arrays size	27%
L1-norm array calculation	448x448 arrays size	14%
DFT	120 N-Point	18%

thus leading to more energy consumption that the baseline system. However, as the task data sets increase, all micro-benchmarks are executed more efficiently to the 7-slave MPSoC, thus leading to less energy consumption. As we can see from Table II, under the largest task size, in the cascaded multiplications, L1-norm calculation and DFT benchmarks, the 7-slave MPSoC, consume 27%, 14% and 18% of the baseline system energy respectively.

Comparison to related work: From the above experiments, we can conclude that our proposed runtime support for reconfigurable MPSoCs has the potential to provide significant performance and energy consumption benefits compared to single-core systems. However, in order to exploit these benefits, developers must divide the application into tasks with workload size enough to "keeps busy" each slave processor, in order to overlap the runtime task scheduling overhead.

As it was discussed in Section II, most of the proposed runtime frameworks target advanced systems with contemporary CPUs, optionally coupled with parallel processing platforms, such as FPGAs and GPUs. Most works support a task-based programming model with many code-annotation clauses and advanced techniques for internal task scheduling. In our work we adopt a similar approach; developers partition their application into tasks, and annotate their code with certain pragmas. Unlike other approaches like in [16], where the application needs to be described using the Polyhedral Process Networks, we believe that code-annotation with simple key-words offers increased productivity on application development. Consequently, we expect that there should be minimal effort to port already task-partitioned applications for execution to our system; basically annotate all application tasks with our runtime's pragma key words.

Furthermore, in contrast to [11], [16], [18], an important

benefit of our work is that it does not require that slave cores support interrupts. This fact considerably relaxes the slave cores and hence overall system complexity, thus enhancing the runtime general purpose applicability. Finally, in our current runtime implementation we utilize a simple task scheduling policy, in contrast to advanced runtime systems. Our modular runtime design though, provides an easy way to include more predefined task scheduling policies. Furthermore, it allows developers to import their own ones, thus offering an additional level of runtime customization, in order to fit the application requirements.

Another benefit of our runtime system is that it does not require a complete embedded OS. As it was mentioned in Section II, in [14], [15], the authors utilize the uCLinux embedded OS, which although specifically targets embedded platforms, still introduces an overhead in terms of performance and resources utilization. In contrast, as we described in Section IV, our *master runtime partition* requires a simple kernel for scheduling its two main modules, the *master runtime* and *recover runtime*, while there is no OS running on any worker processor. Combining the fact that our runtime may be easily ported to any shared-memory *heterogeneous* MPSoC, we believe that it provides an efficient and lightweight solution for rapid development of parallel applications running on embedded MPSoCs.

V. Conclusions

In this paper we have presented a runtime system that can efficiently support shared memory MPSoCs, which are based on a master processor that schedules tasks to slave ones. As it was described, our proposal requires that developers add simple code annotations to the original application software before the tasks that will be executed to slave processors. Moreover, we developed a Java C-to-C compiler that generates C source files for the master and slave processors, which can be used by vendor-specific tools and generate a complete reconfigurable MPSoC with runtime support. Based on our experimental results from our FPGA-based system, we conclude that our work has the potential to provide significant performance and energy consumption saving benefits, as long as the developer carefully partitions the original application code into "coarse-grained" tasks.

References

[1] Silicon Graphics International, "SGI UV: Big Brain for No-Limit Computing."

[2] Convey Computer, "Convey MX Series - Architectural Overview."

[3] OpenMP.org, "OpenMP Application Program Interface Version 4.0 - RC1," November 2012.

[4] Alejandro Duran et al., "OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures," in *Parallel Processing Letter*, vol. 21, June 2011, pp. 173–193.

[5] George Tzenakis et al., "BDDT: Block-level Dynamic Dependence Analysis for Deterministic Task-Based Parallelism," February 2012.

[6] Pieter Bellens et al., "CellSs: a Programming Model for the Cell BE Architecture," in *ACM/IEEE Conference on Supercomputing*, 2006, pp. 85–95.

[7] Enric Tejedor et al., "ClusterSs: A Task-Based Programming Model for Clusters," in *High Performance Distributed Computing*, 2011, pp. 267–268.

[8] Vincent Nollet et al., "A Safari Through the MPSoC Run-Time Management Jungle," in *Journal of Signal Processing Systems*, vol. 60, August 2010, pp. 251–268.

[9] Arun Patel et al., "A Scalable FPGA-based Multiprocessor," in *FCCM*, April 2006, pp. 111–120.

[10] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.0," September 2012.

[11] Antonino Tumeo et al., "A Dual-Priority Real-Time Multiprocessor System on FPGA for Automotive Applications," in *DATE*, 2008, pp. 1039–1044.

[12] Hayden KwokHay So et al., "A Unified Hardware-Software Runtime Environment for FPGA-Based RCs using BORPH," in *ISSS+CODES*, October 2006, pp. 259–264.

[13] Baskiyar Sanjeev, Natarajan Meghanathan, "A Survey of Contemporary Real-time Operating Systems," in *Informatica-LJUBLJANA*, October 2005, pp. 233–240.

[14] Stefano Bertozzi et al., "Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study," in *Design, Automation and Test in Europe*, March 2006, pp. 1–6.

[15] Andrea Acquaviva et al., "Assessing TaskMigration Impact on Embedded Soft Real-Time StreamingMultimedia Applications," in *EURASIP Journal on Embedded Systems*, vol. 2008, January 2008.

[16] Paolo Meloni et al., "System Adaptivity and Fault-tolerance in NoC-Based MPSoCs, the MADNESS Project Approach," in *Digital System Design*, September 2012, pp. 517–524.

[17] Sven Verdoolaege et al., "pn: A Tool for Improved Derivation of Process Networks," in *EURASIP Journal on Embedded Systems*, vol. 2007, April 2007.

[18] Gianluca Durelli et al., "Automatic run-time manager generation for reconfigurable MPSoC architectures," in *ReCoSoC*, July 2012, pp. 1–8.

[19] Taho Dorta et al., "Reconfigurable Multiprocessor Systems: A Review," in *International Journal of Reconfigurable Computing*, 2010.

[20] Yongji Zhou et al., "Real-Time Modeling of Wheel-Rail Contact Laws with System-On-Chip," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, May 2010, pp. 672–684.

[21] James Dykes et al., "A Multiprocessor System-on-Chip Implementation of a Laser-based Transparency Meter on an FPGA," in *FPT*, December 2007, pp. 373–376.

[22] Ravi Kiran Karanam et al., "A Stream Chip-Multiprocessor for Bioinformatics," in *ACM SIGARCH Computer Architecture News*, vol. 36, May 2008, pp. 2–9.

[23] George Mplemenos, Ioannis Papaefstathiou, "MPLEM: An 80-processor FPGA Based Multiprocessor System," in *FCCM*, April 2008, pp. 273–274.

[24] Dimitris Theodoropoulos et al., "A 3D-Audio Reconfigurable Processor," in *ACM/SIGDA FPGA*, February 2010, pp. 107–110.

[25] Cédric Augonnet et al., "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, " in *Euro-Par*, July 2012, pp. 187–198.

[26] Kaushik Ravindran et al., "An FPGA-based Soft Multiprocessor System for IPV4 Packet Forwarding," in *FPL*, August 2005, pp. 487 – 492.