Efficient Task Scheduling for Runtime Reconfigurable Systems

Mahmood Fazlali[1,2], Mojtaba Sabeghi[2], Ali Zakerolhosseini[1] and Koen Bertels[2]

[1]Department of Computer Engineering, Shahid Beheshti University G.C, Tehran, Iran,
[2]Computer Engineering Lab, Delft University of Technology, Delft, The Netherlands
{m.fazlali, sabeghi}@tudelft.nl, a-zakery@sbu.ac.ir, k.l.m.bertels@its.tudelft.nl,

Abstract. Recent research indicates the promising performance of employing reconfigurable systems to accelerate multimedia and communication applications. Nonetheless, they are yet to be widely adopted. One reason is the lack of efficient operating system support for these platforms. In this paper, we address the problem of runtime task scheduling as a main part of the operating systems. To do so, a new task replacement parameter, called *Time-Improvement*, is proposed for compiler assisted scheduling algorithms. In contrast with most related approach, we validate our approach using real application workload obtained from an application for multimedia test remotely taken by students. The proposed on-line task scheduling algorithm outperforms previous algorithms and accelerates task execution from 4% up to 20 percent.

Keywords: FPGA, Reconfigurable Computing, Runtime Task Scheduling, Operating System.

1 Introduction

Reconfigurable computing is a promising technology to meet ever-increasing computational demands by leveraging the flexibility and the high degree of parallelism offered by reconfigurable fabrics, such as Field Programmable Gate Arrays (FPGAs) [1]. Many multimedia applications include computational intensive kernels that can be accelerated by reconfigurable computing [2]. This is common not only for domain-specific applications, but also for many applications in general-purpose computing and even embedded systems domains (e.g. PDAs and cellular phones). In view of the fact that current FPGAs have millions of gates, it is now feasible to consider the possibility of serving several applications on a high performance reconfigurable machine. Therefore, the system should be able to share the FPGA resources among the tasks within a single application or even among different applications. This is possible via runtime Reconfiguration (RTR) of the FPGA and appropriate scheduling of tasks. A well-known disadvantage of reconfigurable systems is that the reconfiguration latency may generate significant overheads. Nonetheless, one of the challenges is to limit the configuration overhead caused by such reconfiguration [3] and managing the reconfigurable resources.

The tasks in a single application environment can be scheduled efficiently by partitioning and design-time scheduling of the tasks [4, 5]. However, such totally predictable application schedules form only a small subset of the total class of applications. Hence, for multi-tasking environments, a runtime scheduler is required and the reconfigurable resources should be managed at runtime [6-7]. There are two major solutions for runtime scheduling the tasks in reconfigurable computers. The former is the scheduler that uses past runtime information to predict the future [8-9]. When the Operating System (OS) performs a context switch, a different application will be executed which have different task needs. Therefore, in multitasking systems the recent past behavior may not predict the near future efficiently [10]. The latter solution is using the design-time profiling information to schedule the tasks at runtime. To do so, a compiler assisted scheduling algorithms were proposed in [11] that uses *Configuration Call Graph* (*CCG*) to schedule the tasks. In this paper we have followed this approach by presenting a new scheduling algorithm.

Although the obtained results in previous related studies indicate some speed-up in the system, they are evaluated using randomly generated task sets which do not capture the real specifications of modern multimedia applications. This is a drawback to compare scheduling algorithms and in this paper we present a real application workload to remove this weakness.

The main contributions of this paper are:

The definition of a new replacement parameter for compiler assisted task scheduling algorithm using a modified *CCG.*

The use of a real application workload to validate the runtime task scheduling algorithm

In addition, we implemented the scheduling algorithms presented in [9] and [11] to compare the results. The rest of the paper is organized as follows. First, the state-of-the-art is reviewed in the next section. Afterwards, Section 2 defines the problem and its background. In Section 4 we discuss an application scenario to motivate the use of runtime scheduling in a multi tasking environments. In section 5 we present the scheduling algorithm. Section 6 includes the proposed workload for evaluation. Section 7 shows the experimental results. Ultimately Section 8 concludes this article.

2 Related Work

Many researchers have presented techniques for managing multitasking reconfigurable systems in which the tasks are assigned at runtime to Reconfigurable Processors (RPs). Two main goals of such mapping is "minimizing the execution time of the tasks" and " virtualising the hardware allocation in a multitasking environment". A number of articles present a general approach to extend a runtime environment system or an OS with the capabilities to manage hardware resources [12-20].

In [12] the authors present a virtualization layer that lowers the interfacing complexity and improves program portability. The layer shifts the burden of moving data between the General Purpose Processor (GPP) and RP from the programmer to the OS. In [13] a virtualization layer is presented to manage RPs in a multitasking environment. The virtualization layer decouples the process of software development from hardware design which results in the software to be independent of the underlying reconfigurable hardware. In [14] Taher and Ghazawi formulate the virtual configuration management technique which does so by discovering and exploiting spatial and temporal processing locality at runtime for RCs. The developed techniques extend existing memory management strategies to reconfigurable platforms and augment them with data mining concepts using association rule mining.

Other approaches have developed runtime support that can manage reconfigurable resources transparently and with a good performance. For instance, in [15] Wigley and Kearney review the services needed for reconfigurable OS and present in [16] a prototype operating system known as ReConfigME. ReConfigME includes details on the selected platform and the detailed implementation. In [17] and [18] the ReconOS is presented which extends the concept of multithreaded programming to reconfigurable logic. ReconOS aims to provide hardware cores with the same services as the software threads of contemporary operating systems, thereby transferring the flexibility, portability and reusability of the established multithreaded programming model from software to reconfigurable hardware. In [19] Nollet et al. propose a distributed OS support for inter-task communications. Finally in [20] Hayden et al. present a LINUX-based OS whose interface has been extended in order to deal with hardware processes.

Other objectives when defining new reconfigurable OS functionality are a reduction of the FPGA fragmentation, minimizing the task rejection rate or communication between the tasks. These approaches depend on placement strategies used in the system. Among the previous work, in [21, 22] Walder, Platzner et al. present some techniques to manage and schedule the execution of task graphs in a one-dimensional, block-partitioned, reconfigurable device. One limitation of their approach is that they assume non-rectangular tasks which is not a realistic assumption for the current technology. In [6] Handa and Vemuri propose an integrated online scheduling and dynamic placement methodology to manage the empty area of an FPGA as a list of maximal empty rectangles to reduce the fragmentation of the FPGA. Researchers in [23] present a task scheduler model and correlative algorithm for scheduling software, hardware and hybrid tasks. Their scheduler combines task allocation and task placement with task migration. However, we do not consider task migration and dynamic placement of the tasks to be more realistic.

In [24] Pellizzoni and Caccamo propose a pseudo-optimal allocation algorithm and a relocation scheme for reloadable tasks but they have employed dynamic allocation for real time tasks. In [25] and [26] Ahmadnia et al. use methods from algorithmic and mathematical optimization to present algorithms for placing, scheduling, and defragmenting tasks on FPGAs. Taking communication between modules into account, they present strategies to minimize communication overhead. However, these approaches are not compatible with our scheduler because we do not consider dynamic placement in our work as they are poorly (if ever) supported by today's platforms. Moreover, we do not consider task dependency between the tasks.

In [27-29] Daniel Mozos et al. have presented hardware task graph schedulers for multitasking systems to reduce the configuration overhead and increase the system speed-up. However, they have focused on a hardware-implemented scheduler because executing complex scheduling algorithms at runtime may generate an excessive overhead. Similarly, [30] presents a hardware-supported task scheduling mechanism on dynamically reconfigurable SoC architectures that is especially compatible with embedded systems.

A closely related approach is by Fu et all in [9] where they employ the past configuration information in the scheduler to predict which (hardware) task in a program will be most beneficial in the near future. In [11], Sabeghi et.al use design-time profiling information to schedule the tasks at runtime. They use a *Configuration Call Graph* (*CCG*) to replace the tasks on reconfigurable hardware. In this paper, we have improved this approach by presenting a time-improvement parameter in *CCG*. Moreover, we have implemented these algorithms beside ours to evaluate the results.

3 Background Overview

The runtime scheduler is at the core of the run-time environment (or operating system) managing task execution in multitasking reconfigurable system. Therefore, we need an efficient scheduler to perform both managing and accelerating of hardware tasks in reconfigurable systems. The main challenge addressed in this paper is to efficiently handle the task assignment mapped on the RP or GPP. In our system, we assume that the set of applications has been analyzed at design-time in order to obtain the *CCG*s. For our purposes, we have analyzed multimedia tests in [31] to create the *CCG*s for the proposed application workload in this paper which will be explained latter.

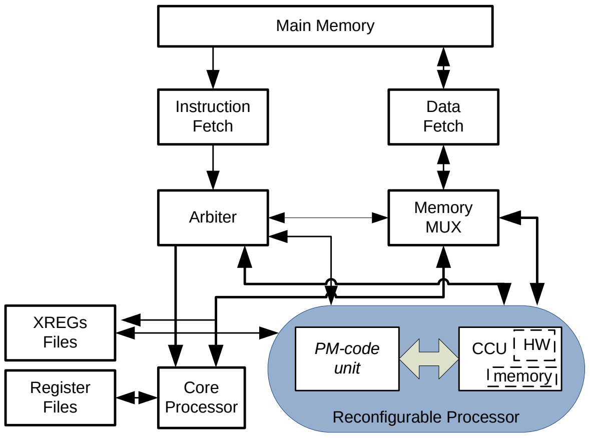


Fig. 1. Molen Hardware Organization

The runtime environment presented in [13] is in fact a virtualized interface, which decides how to allocate tasks to RPs at runtime. Two important modules in the runtime environment are the scheduler and the profiler. The scheduler cooperates with the profiler that continually tracks the application behavior and records statistics obtained from the execution of the algorithm [13]. The profiler can update *CCG*s at runtime. However; this update is done in parallel with the scheduling of the various tasks as performed by the runtime environment scheduler. Such a scheduler needs to take into account whether such a task may have been configured previously and thus may still be available or may have to be configured at runtime again.

Our target system architecture is based on the Molen polymorphic reconfigurable processor [32]. Figure 1 presents this Molen organization which is based on the tightly coupled processor-coprocessor architectural paradigm. In this system, less CPU intensive tasks as well as control of the tasks are assigned to the GPP, whereas computing intensive tasks are accelerated using the RPs. Moreover, within the Molen concept, the GPP controls the execution and reconfiguration of RPs. The Molen hardware organization was explained in [32]. The tasks are tried to be executed on RPs if possible. If not, they will be executed on GPP. In our research, we divide the entire area of the FPGA into a set of tiles for each hardware implementation. The hardware implementation should be loaded to the same set of tiles to configure RPs. We develop the scheduler to assign tasks to these tiles at runtime. We use the fixed tiles on the FPGA and the tiles have some common parts. For example one tile is split to create two smaller tiles. Therefore, our partitioning is somehow between fixed partitioning and dynamic partitioning. We consider that tasks are independent meaning that there is no inter-task communication. In order to configure tasks on RPs simultaneously, it is likely that several configurations will be needed. However, current reconfigurable platforms only include one reconfiguration controller. Hence, they can only carry out one reconfiguration at a time. Our scheduler considers this limitation when configures the tasks.

The programmer's interface in the Molen programming paradigm consists of a series of instructions of which 'SET' and 'EXECUTE' are the most important. These instructions abstract away the underlying hardware complexity for the programmers and provide both compiler and runtime support to efficiently use the underlying hardware. " SET and EXECUTE " respectively load and execute a hardware implementation on the reconfigurable processor. The runtime scheduler considers the EXECUTE instruction as the starting point of the tasks but needs to take into account the latency of the configuration, which is represented by the SET instruction. The compiler will perform a first schedule of the different SET and EXECUTE instructions resulting in a Configuration Call Graph (CCG). However, the runtime system can override any of the scheduling decisions made by the compiler [33].

4 Application Scenario

Our application scenario is based on a multimedia internet based testing application. It is similar to the TOEFL iBT exam. In such a test, there are several test takers (applicants) connected to the exam server. There is a separate process on the server for each test taker and this process has to send the questions containing multimedia features such as voice, video and pictures to test takers. Each test taker might use his own machine to connect to the server therefore, there are different machines with different computing powers connected to the server. As a result, the server must send the question in a format which can be easily decoded by the clients. Furthermore to ensure the security, the questions have to be encrypted. In the following paragraph, we present an overview of the test structure.

In the Listening tests, the server should send image and voice files for each question to the client. Therefore, the server encodes image and voice files. Afterwards, the test question is sent to the client. Whenever a user answers the test, the test answer should be sent to the server. To have a secure exam, the client encrypts the test answer file and sends it to the server. In the Speaking tests, similar to the Listening tests, image and voice files inside the test question are to be sent to the client. Again, they are encoded and encrypted by the server before being sent to the client. In the Reading tests, the server sends the simple test to the client and receives the test answer files that are encrypted by the client. For these different tests, and given that many simultaneous users will perform similar operations, the encoding and encrypting of the files can be accelerated by mapping them on reconfigurable fabric.

5 Runtime Task Scheduler

The runtime scheduler is the main part of a multi-tasking reconfigurable machine that dynamically binds tasks to the *GPP* or *RP*. Although the software tasks can be preempted, the hardware tasks cannot be preempted. Also the scheduler decides which task should be executed on the *RP* at which time. This is done based on dynamic conditions of the system.

For example in our application scenario, the clients' requests for different tasks are not known beforehand by the server. This means that the server only knows the requests when the client issues them. Furthermore, the number of clients and as a result the number of requests may change during the system run. These are the dynamic conditions in the system. The goal of the scheduler is to allocate *RPs* to the tasks that provide the maximal speed-up. To this end, the scheduler employs a replacement policy to determine which task on *RP* should be replaced whenever new task comes. Of course, it is preferable to take out the task that will contribute the least to future performance speed-up.
Fig.2.a shows the runtime environment [34]. The JIT compiler in this figure can be used to compile the tasks for which there is no implementation in the library. The compiler converts the binary code to a bitstream. The transformer replaces the software implementation of the task with a call to the hardware implementation, whenever the scheduler decides to run a task in hardware. The profiler continually tracks the application behavior and records statistics obtained from the execution of the algorithm [34].The main part of the runtime environment is the scheduler that replaces the tasks. The scheduler employs the *CCG* to look at future configuration requests to extract useful information which can be used as the replacement decision parameter. The *CCG* is made by the design-time tool-chain. As illustrated in Fig.2-b the nodes of the *CCG* are of three types. The executable node represents a computational intensive task which should be executed on the RP. Edges of the *CCG* represent the dependencies between the configurations within the application. The parallel execution is represented by the *AND*-node in the *CCG* specifying that all successor nodes can be executed simultaneously. The *OR*-nodes in the *CCG* specify that only one of their successors has to be executed. The output edges of the *OR* nodes are weighted with the probabilities P_i of the execution of the corresponding successors. These probabilities are computed based on run time profiling of the application. The compiler is then responsible for creating the entire *CCG* [11].At runtime, the scheduler employs multiple *CCGs* to schedule the task in multitasking systems. To do so, it combines *CCG*s and uses a task replacement parameter to schedule the tasks.

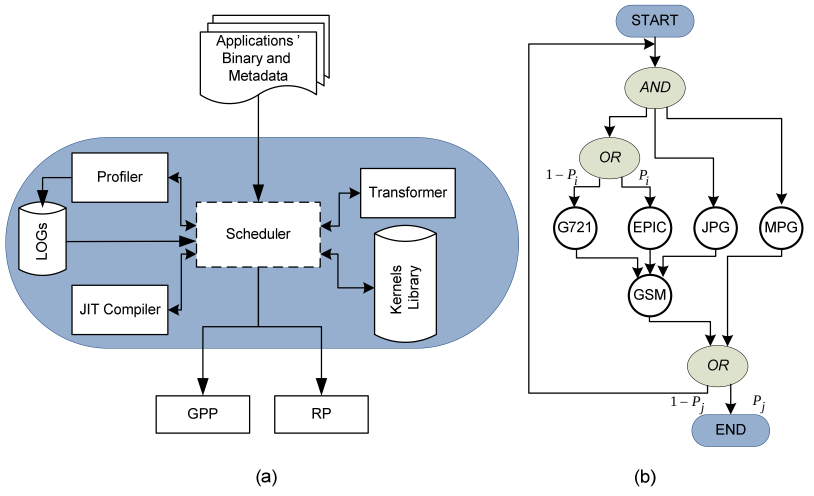


Fig. 2. (a) The runtime environment and, (b) Configuration Call Graph

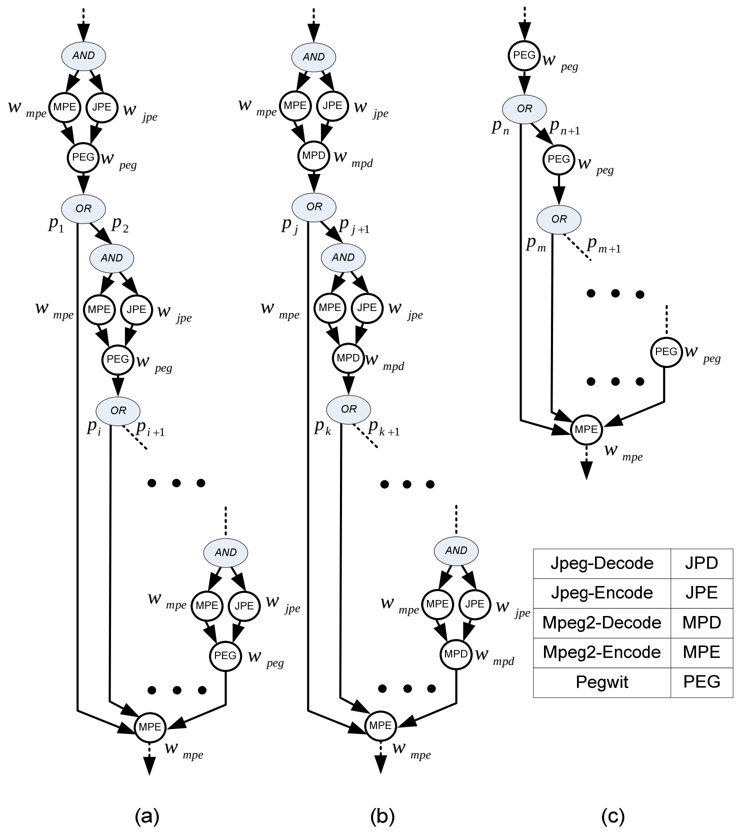
5-1 Time-Improvement Parameter

In the scheduling mechanism presented in [11], task replacement is based on the distance-to-the-next call parameter. This implies that those tasks are removed which will be used furthest away in the future. Although this decision rule results in less replacement and consequently less configuration overhead, it cannot maximize the total speed-up which is the main purpose of employing the reconfigurable computer. In this paper, we propose the *Time-Improvement* heuristic which will be shown to provide better performance. The Time-Improvement heuristic is defined by two parameters: the first is the reduction-in-task-execution time which is obtained by comparing the hardware execution time of a task on FPGA and the software execution time on the general purpose processor. The second parameter is called distance-to-next-call.

Reference [11] describes how this parameter is calculated and what is the role of the OR nodes in the calculation. Our scheduling algorithm employs both of these parameters together.

To this purpose, we have introduced a modified *CCG*, similar to the *CCG* defined in [11], but the nodes are weighted. The weights of a node $W_{task} = \{w_1, \ldots w_n\}$ indicate the reduction in the task execution time parameter for a task T_i. Assuming that multiple implementations of the same task are available, w_1 represents the execution time reduction of the first implementation of the task and w_n is the execution time reduction of the n^{th} implementation of the task. W_is are calculated for the task in off-line. At runtime, they will be used in our Time-Improvement heuristic.

Fig.3 shows a sample *CCG* type used in this paper for three multimedia test applications. In these *CCG*s, the executable node (e.g. *MPE*, *MPD*) represents a computational intensive task. In the application corresponding to the *CCG* Fig.3-a, image and voice should be encoded simultaneously, represented by the *AND* node that precedes the nodes MPE and JPE. The *OR*-node following the PEG node shows the probability of selecting the next same subsequent node of the application or finishing this application.



Jpeg-Decode	JPD
Jpeg-Encode	JPE
Mpeg2-Decode	MPD
Mpeg2-Encode	MPE
Pegwit	PEG

Fig. 3. Modified Configuration Call Graph (*CCG*)

The Time-Improvement parameter, T_i (d_i,w_i) is a two-parameter heuristic that shows possibility of the acceleration by the task T_i in the future. d_i represents the probable number of task calls, between the current execution point and the next task similar to T_i in the breadth-first traversal of the *CCG*. However, the depth of the breadth-first traversing the *CCG* in each step of the algorithm is limited to accelerate the scheduler. Also, w_i is the possible execution time reduction of the next task similar to T_i. w_i is based on which task implementation is chosen at runtime. In Fig.3a, the distance-to-next-call for the first task JPE is 3 which means there are three numbers of task calls between the current JPE and the next JPE task in the breadth-first traverse of the *CCG*. In our proposed scheduling algorithm, the heuristic to replace a task $T_i = T(d_i,w_i)$ with the current tasks $T_j = T(d_j,w_j)$: $_{j=1,..,m}$ on FPGA is:

$$d_i < d_j \text{ and } w_i - t_c > w_j \quad {}_{for\ j=1,..,m} \Leftrightarrow T(d_i,w_i) > T(d_j,w_j) \qquad (1)$$

In this equation T_j are the replaced tasks on the FPGA when the incoming task should be configured instead of them, and m is the number of implementation for T_j. Configuration is not necessary for the tasks which exist on the FPGA but it should be performed for the coming tasks. Therefore in equation.1, we subtract configuration time (t_c) from w_i. This equation should be interpreted as follows: considering Time-Improvement as a decision heuristic, the task to be replace should have the least possible execution time reduction in future and also be used furthest ahead in future.

5-2 The Scheduling Algorithm

We employ the Time-Improvement parameter to create the scheduling algorithm. There are two types of resources to execute the tasks, *GPP* and *RP*. At each scheduling point, if there is an idle configured implementation on the FPGA which can perform the task, it will be used without the configuration overhead. If none of the configured implementations can perform the task, we have to replace at least one of them with an implementation of the current task. The scheduling algorithm is shown in listing 1.

Let us assume that at a certain point, the scheduler has to decide about whether or not load a hardware task to the FPGA. Furthermore, let us assume that there are different hardware implementations in the Task-Implementation list matching this particular task. The **Configured-Task-list** contains the information of all the task implementations already configured on the FPGA. First, the scheduler checks each entry in the Configured-Task-list. For these tasks there is no need for reconfiguration and, the hardware execution can start right away if it matches input task and if the tile occupied by a candidate implementation is not used by another existing hardware task. However, there might be more than one task implementation on FPGA. In this way, in a Loop (line-2) all the configured task on FPGA are searched to find an implementation which has the fastest execution time.

If no configured task is found, the scheduler has to choose either to replace one or more of the currently configured tasks on FPGA with one of the task in the **Task-Implementation list**. First of all, the available physical locations (tiles) are checked for free space. If there is no free space, the input task is either executed in software or a mapped task is selected for replacement. This can be done using the replacement policy. In line 13 of the algorithm, the task implementations in Task-Implementation list are sorted from the highest-speed implementation to the slowest one. Afterwards, the replacement decision is being taken in a loop (line 14). In this way, the highest-speed implementation is removed from the Task-Implementation list (line 16) and all configured tasks on FPGA which have overlap with this implementation are added to the **Evicted-Task list**. We use the Time-Improvement as our decision heuristic to replace the configuration. As described before, the heuristic fires if:

- The reduction-in-task-execution time of input-task is more than the reduction-in-task-execution time of the tasks in Evicted-Task list.
- The distance-to-next-call of input task is less than the distance-to-next-call of each task in the Evicted-Task list.

Otherwise, the incoming task has to be executed in software. This means that if there is only one task in Evicted-Task-List which is more likely to be reused and has more reduction-in-task-execution time, the algorithm will not configure the incoming task to execute by HW. Hence, the cost of replacing each evicted task has been considered in our algorithm.

The overhead of the algorithm in listing 1 is very dependent on the number of implementations for each number of evicted tasks and much less on the probable number of task calls and reduction-in-task-execution time. Reduction-in-task-execution time can be calculated off-line. Of course, combing the data (distance-to-next-call and reduction-in-task-execution time) from different *CCG*s has also to be done at runtime.

Listing 1. The Scheduling Procedure to schedule input-task at a certain scheduling point

TimeImprove (Input-Task, *CCGs*): Function to calculate Time-Improvement in CCGs.

```
TimeImprovment Scheduling procedure (Input: Input-Task, Configuration Call Graphs (CCGs))
Assume {List: Configured-Task { },   /*List of implementations for the configured task on FPGA*/
              Task-Implementations { }, /*List of all implementations for the input-task */
              Evicted-Task { } /*List of tasks that should be evicted*/
       }
Begin
  For m=1 to number of tasks in  Configured_Task_list
    If (IsConfigured (m) and NOT IsBusy(m)) Then
      Configured=1;
      If (m. ExecutionTime  < FastestImplementation. ExecutionTime)
        FastestImplementation = m; /*Finding the fastest implementation on FPGA to execute the input-task */
    EndIf;
  EndFor;
  If (Configured=1) Then
    Execute (m); /*Executing input-task employing the fastest implementation on the FPGA and finishing the procedure*/
    Return;
  EndIf
  InputTask.Implementations.sort(); /* Sorting the task-implementation list based on the reduction in execution time for the implementations */
  Do
    NextImplementation=0;
    FastestImplementation = InputTask.Implementations.First(); /* Removing the highest speed implementation*/

    Evicted_Task_list  = Overlap (FastestImplementation, Configured_Task_list); /* the configured tasks on FPGA (in Configured-Task
      list) that have physical overlap with High speed-Implementation are added to the Evicted-Task list */
    For i=1 to number of tasks in Evicted_Task_ list
      If   (TimeImprove(Evicted-Task {i}, CCGs) > TimeImprove (FastestImplementation, CCGs)) Then
        Next-Implementation=1;
    EndFor
    If (Next-Implementation=0) Then
      Configure (FastestImplementation); /*Configuring the highest speed implementation on the FPGA*/
      Execute (FastestImplementation); /*Executing Input-Task employing the highest speed implementation on the FPGA*/
      Return;
    EndIf
  While (InputTask.Implementations.IsNotEmpty())
End
```

6 Workload for Evaluation

As mentioned in section 4, the workload is obtained from an interactive multimedia internet based testing application which can serve simultaneously a large number of applicants [31]. Through profiling the exam server, we have identified eight multimedia applications that consume most of the server computation time and are described in the following paragraphs.

6-1 Workloads kernels

The profiled applications include:

- Jpeg-Encoder and Jpeg-Decoder: Jpeg is a standardized compression method for the images. Jpeg is lossy compression, meaning that the output image is not exactly identical to the input image. Two kernels are derived from the Jpeg; Jpeg-Encoder does image compression and Jpeg-Decoder, which does decompression.

- Epic-Encoder and Epic-Decoder: The compression algorithms which are based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. Extremely fast decoding of epic makes it suitable to be employed for portable embedded systems.
- Mpeg2-Encoder and Mpeg2-Decoder: Mpeg2 is the standard for digital video transmission.
- G.721: is a standard for speech codec that uses the Adaptive Differential Pulse Code Modulation (ADPCM) method and provides toll quality audio at 32 Kbps.
- Pegwit: A program for public key encryption and authentication. It uses an elliptic curve over GF(2255), SHA1 for hashing, and the symmetric square block cipher.

In order to implement the profiled applications, we use the C code of the programs in the mediabench [35]. The characteristics of the kernels in the mediabench makes them suitable for mapping on the RP in reconfigurable computer [36][37]. Initially the programs should be converted to an intermediate representation. This way, each program is compiled using the GCC compiler [38], and is profiled to determine which kernels contributed most to the overall program execution time. This way, the kernels in a program are found. For each such kernel, a DFG is generated from the kernel body of the RTL code (intermediated representation in GCC). Using RTL instead of machine instructions permitted us to extract the program code after machine-independent code optimizations, but before register allocation and machine-dependent optimizations. Moreover, whenever possible, procedure integration (automatic in-lining) is applied. The section of the application code corresponding to a CDFG can contain control constructions, such as "if-then", "if-then-else", and "switch". For simplicity, we do not handle nested kernels. The DFGs are generated using a technique based on if-conversion and using condition bit vectors.
If we have a variety of implementations for each task, the runtime scheduler can decide which implementation is suitable to be configured on the reconfigurable fabric. We assume we have three different implementations per task besides their software version. To this end, we apply three synthesis methods to the DFGs of the tasks. These methods are based on the techniques presented in [39], [37] and [40]. We respectively refer to these techniques as the **Conventional module, Merged module** and **Advance Merged module**. To create Conventional module the DFGs are synthesized separately by using the conventional synthesizer which creates the datapaths for the input DFGs. For the Merged module and Advance Merged module the input DFGs are combined together to create a merged datapath. There is a difference in speedup and reconfiguration times between these modules. The Advance Merged module has the minimum configuration time but it has the highest task execution time for all eight benchmarks. The Conventional module has the minimal task execution time for each benchmark but the highest configuration times. The configuration time and execution time of the benchmarks in Merged module is situated between the Conventional module and Advance Merged module. On the basis of these numbers, the scheduler can choose the best replacement module.

Table 1. the software execution time and hardware execution times for the tasks

Benchmarks	**software execution time of the task and task configuration time and task execution time**						
	software execution time of the task (*ms*)	**task configuration time via Conventional module (*ms*)**	**task execution time via Conventional module(*ms*)**	**task configuration time via Merged module (*ms*)**	**task execution time via Merged module (*ms*)**	**task configuration time via Advance Merged module (*ms*)**	**task execution time via Advance Merged module(*ms*)**
Epic-Decoder	19.87	11.04	5.98	6.39	8.53	5.82	8.56
Epic-Coder	11.87	4.87	3.99	2.66	4.93	2.49	5.22
Mpeg2-Decoder	77.35	5.83	2.01	4.11	2.34	3.64	2.43
Mpeg2-Ecoder	10.39	7.51	1.19	5.68	1.82	4.87	1.94
G721	42.42	10.6	3.99	6.39	4.23	5.82	4.64
Jpeg-Decoder	68.39	11.72	7.56	9.13	8.11	8.72	8.63
Jpeg-Encoder	169.33	13.78	29.25	11.49	31.98	10.98	35.23
Pegwit	166.06	12.35	34.56	6.47	32.35	5.88	36.34

After obtaining the bitstream of the hardware implementations, their configuration times are calculated as: configuration time = [(size of bit-stream) / (FPGA clock frequency)] [41]. We calculated the configuration time for each hardware kernel on the FPGA XC5VFX30T. Table 1 lists the information about these kernels and their implementations. The hardware execution time of the kernels is calculated for each implementation. The software execution time of the kernels is computed when running on the GPP. From this we can compute the proportion of the acceleration of hardware execution to the software execution.

6-2 Application Workload

In general, each application in the workload is a type of multimedia test such as Reading, Listening, Speaking or Writing in an interactive multimedia internet based testing. Each application includes a number of tasks. Therefore, the application-mix depends on the ordering of the multimedia tests and number of tasks in the application. The system simulates multiple executions of the applications in the multimedia tests. For example, in an application workload there are five Reading tests, six Listening tests, five Speaking tests and two Writing tests. The start times of the applications are different. The application reuse depends on the workload and the similarity between the tasks in the application workload. We obtained the workload by running the examination server in 5 different set-ups (12 applicants (858 tasks), 24 applicants (1660 tasks), 36 applicants (2419 tasks), 48 applicants (3206 tasks) and 60 applicants (4097tasks)). The server's operations have been logged and the workload is extracted from these logs. For each task, it includes the name of the task, the execution time of the task (software only), and the arrival time of the task. The workload is generated per applicant per set-up. So, for each set-up, we exactly know how many applicants there are (number running process in the server), how, when and where the kernels have been called.

7 Evaluation Results

The simulations for a number of test takers are performed in order to evaluate the performance of the proposed scheduling algorithm. We used the same discrete event simulator used in [11] (an extension of the CPUSS CPU scheduling framework [42]). The measures such as number of tasks, software execution time of the tasks, minimum and maximum hardware execution time of the tasks and configuration times of the hardware implementations are depicted in Table 1. In contrast with [11], in this work we employed the real application workload to compare the scheduling algorithms.

Four algorithms have been compared namely, *Past-Frequency*, *Minimum-Distance*, *Future-Frequency* and the proposed algorithm in this paper called *Time-Improvement*. The Past-Frequency has been proposed in [9] with the name Most Frequently Used (MFU). Past-Frequency predicts the future based on the previous information in the application and removes the task which has been used least in the past. The Minimum-Distance and Future-Frequency have been presented in [11]. Minimum-Distance removes the task that will be used furthest in future and in Future-Frequency, replacement candidate is the task which will be used less frequently in the future. Our target architecture is similar to the Molen hardware platform implemented on the Xilinx Virtex series while the runtime environment is the same as [11].

As explained above, we have different setups in our validation experiments where the number of participants and thus the number of tasks vary. Each cell of table 2 contains the execution time of the tasks and the number of executed tasks on RPs in each scheduling algorithm. The first row is the software only execution of the tasks and other rows show the task's execution times for different scheduling algorithms. As illustrated in the table, all algorithms have shorter execution times than software-only execution of the tasks. The Past-Frequency algorithm has not reduced in any noticeable way the execution time of the tasks in set-up$_1$, set-up$_2$ and set-up$_5$. However, for set-up$_3$ and set-up$_4$ Past-Frequency has reduced the tasks execution time quiet considerably. On the other hand, the future-Frequency has worked the same for all of the set-ups. The Time-Improvement scheduling algorithm performs better than Past-Frequency and Future-Frequency algorithms. The reason is that the Time-Improvement replaces RPs at runtime by predicting the performance penalty that occurs due to the replacement of each RP instead of considering the utilization history of the RPs in Past-Frequency or predicting the utilization of RPs using *CCG*s in Future-Frequency. Although the distance-to-next-call parameter can efficiently predict the number of configurations in the future, it cannot minimize the task execution time. The results indicate that for all application workloads, the execution time of the tasks resulting from the Time-Improvement algorithm is lower than for the Minimum-Distance algorithm. It indicates that when only considering the distance-to-next-call parameter, it may evict a task which has high speed-up potential for the future. Therefore, Time-

Improvement reduces the number of reconfiguration besides predicting the performance penalty that occurs due to replacement of each RP. This is possible by employing both distance-to-next-call parameter and reduction-in-task-execution time parameter to replace the tasks in Time-Improvement algorithm.

Table 2. The tasks execution time and number of executed tasks on RPs in each set-up of application for the scheduling algorithms.

Scheduling Algorithms	**Tasks' Execution Time (*ms*)** / **Number of Executed Tasks on RPs**				
	Set-up$_1$: 12applicants (858 tasks)	*Set-up*$_2$: 24applicants (1660 tasks)	*Set-up*$_3$: 36applicants (2419 tasks)	*Set-up*$_4$: 48applicants (3206 tasks)	*Set-up*$_5$: 60applicants (4097 tasks)
Software-only	135654.08	260508.60	381329.44	501860.74	641478.23
	0	0	0	0	0
Past- Frequency	126883.12	226219.53	248866.52	334082.77	618196.87
	194	656	1092	1388	622
Future-Frequency	91230.78	176467.80	267377.39	360294.81	455492.04
	546	978	1380	1718	2276
Minimum-Distance	71727.74	143546.82	216114.94	290748.26	363356.18
	704	1332	1836	2135	3212
Time-Improvement	68648.15	137524.57	202793.31	241565.90	342166.00
	742	1352	1970	2564	3412

The number of executed tasks in each cell of table 2 indicates that in the most cases there is a direct relation between the number of tasks executed on RPs and the execution time of the tasks in each algorithm. If we consider the number of tasks executed on RPs, the results indicate that the proposed Time-Improvement heuristic can execute more number of tasks on RPs than other algorithms. In addition, it attempts to predict the tasks which have more chance to be accelerated by RPs in future and have less performance penalty in task replacement.

Fig.4 shows the obtained speed-up from the hardware execution of the tasks in each scheduling algorithm compared to the software-only execution in the proposed application workloads. The results show that the proposed Time-Improvement algorithm performs better than all algorithms and it achieves 4% for set-up$_1$ to 20% for set-up$_4$ higher speed-up than the best previous scheduling approach we included in our evaluation.

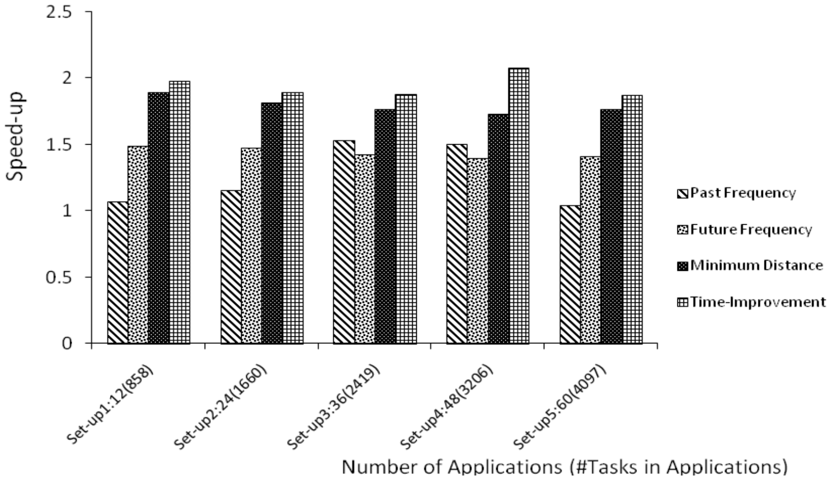


Fig. 4. The tasks execution time in the application set-ups.

The computational complexity of the Time-Improvement algorithm is O(#*Task Implementation* ● #*Evicted Tasks*● *Depth*) where **#Task Implementation** corresponding to the line 29 in the algorithm is the number of task implementations for the incoming tasks**. #Evicted Tasks** corresponds to line 20 shows the number of tasks which should be evicted for configuring the incoming task. The **Depth** means the depth of the breadth-first traversing the *CCG* in each step of the algorithm which was set to 3, representing the three available implementations for each task. Since dynamic partitioning results in significant number of evicted tasks, in our test environment, we do not support dynamic partitioning so, there are up to 4 evicted tasks. Therefore, the time complexity of the Time-Improvement is similar to the Minimum-Distance and Future-Frequency algorithm and does not add significant time-overhead to the system in comparison to the obtained time resulted from accelerating the tasks.

8 Conclusions

This paper presented the Time-Improvement replacement heuristic for runtime task scheduling when managing multi-tasking in reconfigurable computers. To do so, a modified *CCG* is proposed for compiler assisted scheduling algorithms. Employing this heuristic resulted in best overall performance improvement when testing it on a real application workload rather than synthetic workloads. We computed the speed-up of the application running on a server for the scheduling algorithm in this paper and previous scheduling algorithms using the presented workload. The results show that the proposed algorithm outperforms the other scheduling algorithms and achieves a speed-up up from 4% up to 20% more than the best previous scheduling algorithms when validated on different real application scenarios. Although we only considered the Molen machine, comprising a single GPP and a variable number of reconfigurable accelerators, we believe that our scheme can be easily adapted to a hybrid platform consisting of a number of RPs and multiple GPPs by modifying the scheduling algorithm. Even in the presence of a multi-core platform, hardware tasks will still play an important role since they can provide a high degree of parallelism.

ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for the insightful suggestions on improving this paper. Also we thank Iran Telecommunication Research Center (ITRC) and Hartes project for supporting this research in the context of the project T/500/3462 and project EU-IST-035143 respectively.

References

1. K.H. Hayden BORPH: An Operating System for FPGA-Based Reconfigurable Computers. PhD Thesis, University of California, Berkeley, (July 2007).

2. D. Wang, S. Li and Y.Dou, Reconfigurable Computing: Loop Kernel Pipelining Mapping onto Coarse-Grained Reconfigurable Architecture for Data-Intensive Applications, in: Journal of Software (*JSW*), 4(1), 2009, 81-89.

3. K. Compton and S. Hauck, Reconfigurable Computing: A Survey of Systems and Software, in: ACM Computing Surveys (*CSUR*), 34(2), June 2002, 171-210.

4. R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio and D. Sciuto, Partitioning and Scheduling of Task Graphs on Partially Dynamically Reconfigurable FPGA. in: IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems (*TCAD*), 28(5), 2009, 662-675.

5. Z. Gu, M. Yuan and X. He, Optimal Static Task Scheduling on Reconfigurable Hardware Devices Using Model-Checking, in: proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (*RTAS*), WA USA, April 2007, 32-44.

6. M. Handa, R. Vemuri, An Integrated Online Scheduling and Placement Methodology, in: proceedings of the 2004 International Conference on Field-Programmable Technology (*FPT'04*), Brisbane, Australia, December 2004, 444-453.

7. T. Marconi, Y. Lu, K.L.M. Bertels and G. N. Gaydadjiev, Online Hardware Task Scheduling and Placement Algorithm on Partially Reconfigurable Devices. In: proceedings of the International Workshop on Applied Reconfigurable Computing (*ARC08*), London, UK, March 2008, 306 - 311.

8. K. Rupnow and K. Compton, SPY vs. SLY: Runtime Thread-Scheduler Aware Reconfigurable Hardware Allocators, in: proceedings of the 2009 International Conference on Field-Programmable Technology (FPT'09), Sydney Australia, December 2009, 353-356.

9. W. Fu, K.Compton, An Execution Environment for Reconfigurable Computing, in: proceeding of the IEEE Symposium on Field Programmable Custom Computing Machines, Napa Valley CA, USA, April 2005.

10. L. Bauer, M. Shafique, J. Henkel, MinDeg: A Performance-Guided Replacement Policy for Run-Time Reconfigurable Accelerators, in: proceedings of the International Conference on Hardware-Software Co-design and System Synthesis (*CODES+ISSS*), Grenoble, France, October 2009, 335-342.

11. M. Sabeghi, V.M. Sima and K.L.M. Bertels Compiler Assisted Runtime Task Scheduling on a Reconfigurable Computer, in: proceedings of the 19th International Conference on Field Programmable Logic and Applications (*FPL09*), Prague Czech Republic, (August 2009).

12. M. Vuletic, L. Righetti, L. Pozzi, P. Ienne, Operating System Support for Interface Virtualisation of Reconfigurable Coprocessors, in: proceedings of the Proceedings of the Design, Automation and Test in Europe Conference (*DATE04*), Paris, France, February 2004, 748-749.

13. M. Sabeghi and K.L.M. Bertels, Toward a Runtime System for Reconfigurable Computers: A Virtualization Approach, in: proceedings of the Design, Automation and Test in Europe (*DATE09*) Conference, Nice, France, April 2009.

14. Mohamed Taher and Tarek El-Ghazawi, Virtual Configuration Management: A Technique for Partial Runtime Reconfiguration, in: the IEEE Transactions on Computers (*TC*), 58(10), 2009, 1398-1410.

15. G.B. Wigley, D.A. Kearney, Research Issues in Operating Systems for Reconfigurable Computing, in proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (*ERSA*02). Nevada, USA, June 2002, 10–16.

16. G. Wigley, D. Kearney, M. Jasiunas, ReConfigME: A Detailed Implementation of an Operating System for Reconfigurable Computing, in: proceedings of the 20th International Parallel and Distributed Processing Symposium (*IPDPS*), Rhodes Island, Greece, April 2006.

17. E. Lubbers and M. Platzner, ReconOS: An Operating System for Dynamically Reconfigurable Hardware. Book Chapter in Dynamicaly Reconfigurable System, Springer, 2010.

18. E. Lubbers and M. Platzner, Cooperative Multithreading in Dynamically Reconfigurable Systems, in: proceedings of the 19th International Conference on Field Programmable Logic and Applications (*FPL09*), Prague Czech Republic, August 2009, 551-554.

19. V. Nollet, T. Marescaux, D. Verkest, J.Y. Mignolet and S. Vernalde, Operating System Controlled Network On-Chip, in proceedings of the 41st Design Automation Conference (*DAC*), San Diego, California, USA, June 2004, 256–259.

20. H.K. So, R.W. Brodersen, A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers Using BORPH, in: ACM Transactions on Embedded Computing Systems (*TECS*) 7 (2), February 2008.

21. H. Walder, M. Platzner, Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform, in: proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture (*ERSA02*), Las Vegas, CA, June 2002, 24-30.

22. C. Steiger, H. Walder, M. Platzner, L. Thiele, Online Scheduling and Placement of Real-Time Tasks to Partially Reconfigurable Devices, in: Proceedings of the Real-Time Systems Symposium (*RTSS03*), Cancun, Mexico, December 2003, 224-235.

23. L. Liang, X.G. Zhou, Y. Wang, C.L. Peng, Online Hybrid Task Scheduling in Reconfigurable Systems, in: Proceedings of the International Conference on Computer Supported Cooperative Work in Design (*CSCWD07*), Melbourne, Australia, VIC, April 2007, 1072-1077.

24. R. Pellizzoni and M.Caccamo, Real-Time Management of Hardware and Software Tasks for FPGA-Based Embedded Systems, in: the IEEE Transactions on Computers (*TC*), 56(12), 2007 1666-1680.

25. A. Ahmadinia, C. Bobda, D. Koch, M. Majer, J. Teich, Task Scheduling for Heterogeneous Reconfigurable Computers, in: proceedings of the 17th Symposium on Integrated Cicuits and Systems Design (*SBCCI04*), Pernambuco, Brazil, September 2004, 22-27.

26. A. Ahmadinia, J. Angermeier, S. P. Fekete, T. Kamphans, D. Koch, M. Majer, N. Schweer, J. Teich, C. Tessars and J.C. van der Veen, ReCoNodes: Optimization Methods for Module Scheduling and Placement on Reconfigurable Hardware Devices. Book Chapter in Dynamicaly Reconfigurable System, Springer,2010.

27. J. Resano, J. A. Clemente, C. Gonzalez, D. Mozos and F. Catthoor, Efficiently Scheduling Runtime Reconfigurations. In: ACM Transactions on Design Automation of Electronic Systems (*TODAES*), 13(4), September 2008.

28. J.A. Clemente, C. González, J. Resano, D. Mozos, In press as: A Task Graph Execution Manager for Reconfigurable Multi-tasking Systems. Microprocessors and Microsystems, 2010.

29. J.A. Clemente, C. Gonzlez, J. Resano, D. Mozos, A Hardware Task-Graph Scheduler for Reconfigurable Multi-Tasking Systems, in: proceedings of the International Conference on Reconfigurable Computing and FPGAs (*ReConFig08*), Cancun, Mexico, December 2008, 79-84.

30. Z. Pan and B.E. Wells, Hardware Supported Task Scheduling on Dynamically Reconfigurable SoC Architectures, in: the IEEE Transactions on Very Large Scale Integration (*TVLSI*) Systems, 16(11), 1465-1474, 2008.

31. http://precise-testing.com/

32. S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov and E. M. Panainte, The Molen Polymorphic Processor. in: IEEE Transactions on Computers (*TC*), Volume 53, Issue 11, November 2004, 1363- 1375.

33. M. Sabeghi, K.L.M. Bertels, Interfacing Operating Systems and Polymorphic Computing Platforms based on the MOLEN Programming Paradigm. in: 6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture in conjunction with ISCA10, Saint-Malo, France, June 2010, 30-35.

34. M. Sabeghi, H. Mushtaq, K.L.M. Bertels, Runtime Multitasking Support on Reconfigurable Accelerators. in: International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies held within ACM ICS 2010, Tsukuba, Japan, June 2010, 54–59.

35. C. Lee, M. Potkonjak, W S. Mangione, Mediabench: a Tool for Evaluating and Synthesizing Multimedia and Communication Systems, in proceedings of the Annual IEEE/ACM International Symposium on Micro-architecture (*MICRO*), California USA, December 1997, 330–335.

36. M. Fazlali and A. Zakerolhosseini, Rec-Bench: a Tool to Create Benchmark for Reconfigurable Computers, in proceedings of the VI Southern Programmable Logic Conference, (SPL), Porto de Galinhas Bench, Brazil, March, 2010.

37. M. Fazlali, A. Zakerolhosseini, M. Sabeghi, K.L.M. Bertels and G.N. Gaydadjiev, Datapath Configuration Time Reduction for Runtime Reconfigurable Systems, in: proceeding of the proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, (*ERSA09*), Nevada, USA, July 2009, 323-327.

38. http://gcc.gnu.org/onlinedocs . GNU Compiler Collection Internals.

39. Y. D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G. N. Gaydadjiev, Y. Lu and S. Vassiliadis, DWARV. DelftWorkbench Automated Reconfigurable VHDL Generator, in Proceedings of the 17th International Conference on Field Programmable Logic and Applications (*FPL07*), Amsterdam, The Netherlands, August 2007, 697-701.

40. M. Fazlali, A. Zakerolhosseini and G.N. Gaydadjiev, A Modified Merging Approach for Datapath Configuration Time Reduction, in: proceeding of the 6th International Symposium on Applied Reconfigurable Computing (*ARC2010*), Bangkok, Thailand, March 2010, 318-328.

41. M. Rollmann and R. Merker, A Cost Model for Partial Dynamic Reconfiguration, in: proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (*SAMOS*), SAMOS Greece, July 2008, 182-186.

42. Barnett G. CPU Scheduling Simulator, available on www: http://cpuss.codeplex.com/ (2009).