Configuration-Sensitive Process Scheduling for FPGA-Based Computing Platforms

G. Chen, M. Kandemir
Dept. of Computer Science & Engineering
The Pennsylvania State University, USA
{guilchen, kandemir}@cse.psu.edu

U. Sezer
Electrical and Computer Engineering Dept.
University of Wisconsin-Madison, USA
sezer@ece.wisc.edu

Abstract

Reconfigurable computing has become an important part of research in software systems and computer architecture. While prior research on reconfigurable computing have addressed architectural and compilation/programming aspects to some extent, there is still not much consensus on what kind of operating system (OS) support should be provided. In this paper, we focus on OS process scheduler, and demonstrate how it can be customized considering the needs of reconfigurable hardware. Our process scheduler is configuration sensitive, that is, it reuses the current FPGA configuration as much as possible. Our extensive experimental results show that the proposed scheduler is superior to classical scheduling algorithms such First-Come-First-Serve (FCFS) and Shortest Job First (SJF).

1. Introduction

Reconfigurable computing systems have shown the ability to greatly accelerate program execution, thereby providing a high-performance alternative to software-only implementations and a programmable alternative to ASICs. While prior research have addressed architecture design and programming and compilation issues [7,10,1,8,3, 9], there is still not much consensus on what kind of operating system (OS) support should be provided for reconfigurable architectures. Prior OS related work [12] has evaluated different scheduling algorithms on an FPGA based platform, and found that different scheduling algorithms can generate different results. Brebner [2] discusses some basic issues that can impact the construction of an operating system for FPGAs with dynamic reconfiguration capability. He proposes that applications be designed into relocatable cores. Diessel et al [4,5], Gindy et al [6], and Wigley and Kearney [13] discuss high-level OS support for reconfigurable systems.

Obviously, a simple solution to the OS problem in the context of reconfigurable computing is to let the reconfigurable hardware be transparent to the OS. In this case, we do not need to change any OS algorithm/data structure, and reconfigurable hardware is managed by the compiler and/or user application. While this is certainly a viable option in some cases, we believe that making the OS aware of the reconfigurable device could lead to better overall system behavior. There are two major reasons for that:

a. First, if the OS is not aware of the reconfigurable hardware, each application can reconfigure it considering only its own needs, and this may lead to frequent reconfigurations at runtime as the OS scheduler moves from one application to another during the course of execution.

b. Second, the OS can pre-configure the hardware considering the next process to be executed (based on the scheduling information it has), and this can help reduce the time spent in reconfiguration.

Based on these, this paper presents a customized OS scheduling support for an FPGA-based execution environment. Unlike previous work on OS scheduling [12], the scheduling algorithm presented here is *configuration-sensitive*; i.e., it tries to reuse the current configuration as much as possible. Also, we demonstrate that pre-configuration can make a significant performance difference. In addition, we also present a code restructuring strategy, using which we modify process codes to make better reuse of the current FPGA configuration.

Our experiments with this new scheduler and process code restructuring strategy reveal that they can improve performance significantly. Our experimental results also demonstrate that the proposed scheduler is superior to classical scheduling algorithms such First-Come-First-Serve (FCFS) and Shortest Job First (SJF). Based on these results, we recommend the OS designers for embedded systems that employ reconfigurable fabric to customize their process scheduling algorithms.

This paper is structured as follows. The next section describes the reconfigurable environment we target. Section 3 gives details of our approach to process scheduling. Section 4 discusses potential performance

1530-1591/04 $20.00 (c) 2004 IEEE

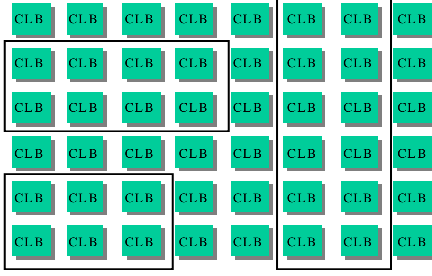

Figure 1. A 6x8 CLB array and regions used by three processes.

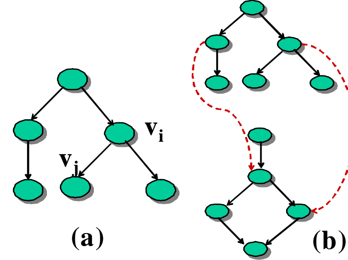

Figure 2. (a) An example STG (note that node vj depends on node vi). (b) Two STGs and data dependences between them (shown as dashed arrows).

benefits due to process code restructuring. Section 5 discusses preconfiguration, an optimization that blends well with our scheduling strategy. Section 6 gives simulation results. Finally, Section 7 provides our conclusions.

2. Reconfigurable architecture and execution environment

In this section, we present the main assumptions that we make about our architecture. We model our reconfigurable system as a rectangular array of configurable logic blocks (say NxM) – called CLBs. We also assume an on-chip interconnect that ties these blocks together. Many commercial FPGAs fit in this description. Each process to be scheduled in this architecture uses a "rectangular portion" of this array. For example, Figure 1 illustrates a 6x8 array and three processes scheduled on it (they occupy spaces of 2x3, 2x4, and 6x2 CLBs). For clarity, we omit from the figure the interconnects between CLBs and the input-output blocks (IOBs). In this environment, when a new incoming process needs to be scheduled, the OS needs to find a (rectangular) space for it in the reconfigurable device (this is called space allocation). This space can be one of the spaces that have already been allocated to some other process that could not run concurrently with this incoming process (this is called *configuration reuse* in this paper). Or alternately, if available, an unused space can be allocated to the incoming process. This decision depends on two main factors: (a) relative execution orders of different processes, and (b) inherent configuration reuse capability between processes. We will discuss these two issues in detail in the next section. *Our main objective is to maximize configuration reuse, that is, we would like to minimize the number of reconfigurations.*

It is important to note here that this model assumes that the configurable device is *partially reconfigurable* (i.e., it can be selectively programmed without a complete reconfiguration [3]) in both row and column directions. This capability does not exist in some of current FPGAs such as Xilinx's Virtex [11]. However, as will be demonstrated later, our approach is also applicable to the cases where the FPGA is reconfigurable only in vertical chip-spanning columns.

It should be mentioned that there can also be a host processor in the system. However, we assume that the portions (from each application/process) that will be executed in the FPGA part have already been decided prior to our scheduling. In the following discussion, when there is no confusion, we use the terms FPGA and reconfigurable/configurable system interchangeably. We are targeting an embedded environment, where processes (tasks) are extracted from a given embedded application. Note that many embedded applications are coded in a modular fashion as multiple co-operating processes.

3. Our approach to scheduling

We represent each process to be scheduled by a subtask graph (henceforth referred to as STG). Each node of this graph represents a process code portion (subtask) that will be executed in a single quantum of time once it gets scheduled. Note that depending on the computation being performed by the node, each node may require a different amount of FPGA space (i.e., when it is mapped to the FPGA, it can demand a different size rectangular region than the others). A directed edge (arrow) from a node $v_i \in$ STG to another node $v_j \in$ STG indicates a data or control dependence from v_i to v_j; that is, v_j cannot be executed before v_i (they can be pipelined in some cases; however, we do not consider pipelining in this paper). Figure 2a depicts a typical STG for a process. In this paper, the jth node of process i is denoted as STG_{ij}.

Since one of the objectives of any FPGA-based system is to maximize FPGA utilization, the OS scheduler should be able to schedule nodes from the STGs of different applications. It should be observed that while one may have the option of executing each process in a strict order (i.e., not starting executing the next one while the previous one is still running), this may not be a very good idea since dependences between the nodes of the STG in question would prevent full utilization of the available FPGA space. Therefore, our approach is oriented towards maximizing FPGA space utilization by parallel execution of multiple processes. Also, since our processes are

extracted from the same application, there might be data dependences between them. For example, Figure 2b shows two STGs and two dependences between (drawn as dashed arrows).

The crucial step in our configuration-aware scheduling strategy is assigning an FPGA space for a process that is about to start executing. Let us first make an important definition:

Context: The context of an FPGA is a list whose each entry indicates the status of an *FPGA region*. An FPGA region can be defined as a quadruple (also referred to as a *context entry*):

[{(s_1,s_2),(e_1,e_2)}, process-id, STG-node-id, config-desc],

where (s_1,s_2) and (e_1,e_2) indicate, respectively, the start and end CLBs for the (rectangular) region in question. Process-id denotes the process that is using that region, and STG-node-id gives the current STG node that is executing on that region. Finally, config-desc describes the current configuration of the region (i.e., what computation is being performed). Note that, even there is no process running in a region, it can still have a valid config-desc (describing the configuration that has been used there the last time).

The OS scheduler has two important tasks:

a. Determining a suitable schedule for the processes to be scheduled.

b. When a process need to start executing, determining where (to which FPGA region(s)) its subtasks should be assigned.

One simple strategy would be separating these two tasks, and first coming up with a schedule and then determining execution region(s) for each process as its turn comes. However, a closer look reveals that these two tasks are in fact highly related. More specifically, the region(s) on which a process can be executed is an important factor that helps us determine whether it should be the next process to execute. Therefore, we propose a configuration-sensitive execution strategy; i.e., at each scheduling step, we select (among the executable process) the one that fits well in one or more regions, without needing reconfiguration (which is typically a costly operation).

Therefore, in the first step of our approach, we determine a *suitability factor* (SF), which indicates how suitable a schedulable process for the current context. We define an SF_i (for process i) at a time instance as the inverse of the *cost* of executing a subtask j of process i (i.e., node STG_{ij} in the corresponding STG) in the current context. This cost can vary depending on whether we need to perform a (partial) reconfiguration or we can reuse a configuration, which is already there in the FPGA device.

Suppose that we are considering STG_{ij} for potential execution on the FPGA (assuming process i is schedulable). Let config-$desc_{ij}$ be the configuration required by STG_{ij} and {(s_1,s_2),(e_1,e_2)}$_{ij}$ be the region requested. Then, we can compute the cost of scheduling STG_{ij} – denoted cost(STG_{ij}) – in the current context as follows:

Case 1. If there exists a context entry of the form [{(s_1,s_2),(e_1,e_2)}, *, *, config-desc], such that {(s_1,s_2), (e_1,e_2)}$_{ij}$ $\leq$ {(s_1,s_2),(e_1,e_2)} and config-$desc_{ij}$ = config-desc, then the cost is assumed to be 0. This represents the case where we have an already configured empty space suitable for process i (Here, a '*' in the second and third slots of the context entry indicate that no active process occupies the rectangular region in question. Also, we use A $\leq$ B to indicate that region B is at least as large as region A).

Case 2. Else, if there exists an entry of the form [{(s_1,s_2),(e_1,e_2)}, *, *, *], such that {(s_1,s_2),(e_1,e_2)}$_{ij}$ $\leq$ {(s_1,s_2),(e_1,e_2)}, then the cost is the time (delay) spent in configuring {(s_1,s_2),(e_1,e_2)}$_{ij}$ amount of FPGA space. This reconfiguration cost is termed as $cost_{reconfig}$({(s_1,s_2), (e_1,e_2)}$_{ij}$) in this paper. Note that this corresponds to the case where we have an empty (i.e., unconfigured) space in the FPGA (Here, a '*' in the last slot of the context entry indicates no configuration for the region in question).

Case 3. Else, if there exists an entry of the form [{(s_1,s_2),(e_1,e_2)}$_{kl}$, k, kl, config-$desc_{kl}$], such that {(s_1,s_2), (e_1,e_2)}$_{ij}$ $\leq$ {(s_1,s_2), (e_1,e_2)}$_{kl}$ and config-$desc_{ij}$ = config-$desc_{kl}$, then process i waits until the lth node of process k is finished. In this case, we have cost(STG_{ij}) = $cost_{wait}$(STG_{kl}), where $cost_{wait}$(STG_{kl}) is the (remaining) time for STG_{kl} to finish.

Case 4. Else, if there exists an entry of the form [{(s_1,s_2), (e_1,e_2)}$_{kl}$, k, kl, config-$desc_{kl}$], such that {(s_1,s_2), (e_1,e_2)}$_{ij}$ $\leq$ {(s_1,s_2), (e_1,e_2)}$_{kl}$ and config-$desc_{ij}$ $\neq$ config-$desc_{kl}$, then process i waits until the lth node of process k is finished. In this case, we have cost(STG_{ij}) = $cost_{wait}$(STG_{kl}) + $cost_{reconfig}$ ({(s_1,s_2), (e_1,e_2)}$_{ij}$), where $cost_{wait}$(STG_{kl}) is the (remaining) time for STG_{kl} to finish. Note that in this case the overall cost has both waiting time and reconfiguration time components (as opposed to the previous case where the cost has only a waiting delay component). If a single STG_{kl} does not provide suggested FPGA space, we consider more such nodes until we have enough space for STG_{ij}.

To sum up, the cost of scheduling STG_{ij} can be 0, or only some reconfiguration latency, or only some waiting latency, or sum of reconfiguration and waiting latencies. Based on this cost model, it is clear that reusing some existing FPGA configuration is very critical if one is to obtain good performance. Our scheduling algorithm is a variant of list scheduling, an algorithm frequently employed by optimizing compilers. In this scheduling, the next STG node to be scheduled (and to be assigned an FPGA space) is the one that leads to minimum cost (considering the four cases above) among all schedulable nodes. In this way, at each step, this greedy heuristic selects the next STG node to be scheduled.

4. Code restructuring

While the configuration-sensitive scheduling approach described above makes maximum configuration reuse within the capacity of list scheduling, it is possible to improve its behavior further by being a bit more careful about how the application is coded. Specifically, the application can be coded in such a way that the scheduler has higher chances for finding a suitable space (with low cost) in the FPGA device.

As an example, let us consider Figure 3. This figure shows an STG for process i (in Figure 3a) and an STG for process j (in Figure 3b). Assuming that process j is dependent on process i (i.e., we cannot execute any node of process j before finishing up all nodes of process i) and the available FPGA space can hold only one node at a time, we see that as we move from process i to process j, we need to reconfigure the FPGA (from addition to multiplication). In fact, executing process i followed by process j requires a minimum of four reconfigurations. However, if it is also possible to code process j as depicted in Figure 3c, we can reduce the number of reconfigurations to three. This small example illustrates that if we consider the way in which processes are coded, it may be possible to reduce the number of reconfigurations.

5. Preconfiguration

As noted earlier one of the main bottlenecks that could prevent us from obtaining good results is the reconfiguration delay. A way of reducing its impact could be using *preconfiguration*; i.e., (partially) reconfiguring the FPGA before such a configuration is actually needed (in an attempt to hide the time spent in reconfiguration).

Our current approach implements the following preconfiguration strategy. If the first case in Section 3 (where we determine the cost of executing STG_{ij}) fails but the second case succeeds (i.e., we have available empty space in the FPGA that needs to be reconfigured), we start reconfiguring FPGA before STG_{ij} is actually scheduled for execution. An important issue here is the time at which reconfiguration starts. Note that if we invoke reconfiguration very early this can lead to poor utilization of FPGA (since this means keeping a portion of the FPGA ready for a subtask that will come much later). Similarly, if reconfiguration is delayed too much, it may not be very beneficial since we may still need to wait for it to complete. In our current implementation, if STG_{ij} is supposed to be executed at time step t, we start reconfiguring the FPGA for it at time step t-1. Consequently, we have $cost(STG_{ij}) = cost_{reconfig}(\{(s_1,s_2),(e_1,e_2)\}) - \Delta_{overlap}$, where $\Delta_{overlap}$ is the time that is saved due to preconfiguration. Similarly, in

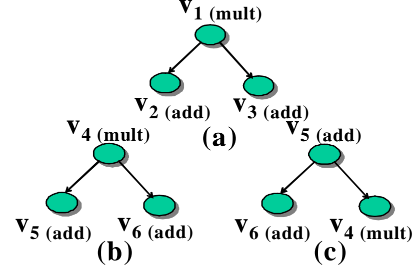

Figure 3. (a) STG for process i. (b) Two alternate STGs for process j.

the fourth case (in Section 3), the preconfiguration for the STG node scheduled for time *t* starts at time *t-1*.

6. Evaluation

6.1. Setup

We compare our approach to two well-known process scheduling techniques: FCFS (First-Come-First-Serve) and SJF (Shortest-Job-First). In FCFS, each arriving process is assigned a time stamp and processes are scheduled according to their stamp values. In other words, the next process to be executed is the one with the earliest arrival time stamp. In comparison, in SJF scheduling, the processes are sorted according to their (estimated) execution times and the process with the shortest time is the one to be executed next. In our implementation of these scheduling algorithms, we worked on a subtask granularity. We postpone the discussion of how our technique can be made preemptive, and how (in that case) it compares to classical preemptive algorithms such as Shortest Remaining Processing Time (SRPT) and Earliest Deadline First (EDF) to a future study.

We implemented our configuration-sensitive process scheduler, FCFS, and SJF, and performed experiments using a custom simulator. Our simulator takes a description for the FPGA device and a process table. Each entry in this table points to a STG and an arrival time stamp. As output, the simulator gives the *average response time* and the *FPGA utilization.* As in [12], the response time of a process i (denoted r_i) is given as $r_i = f_i - a_i$, where f_i is the process's finishing time, and a_i is its arrival time. Then, the average response time is the response time divided by the number of processes in the application. In our simulations, we assumed that reconfiguring one column takes 200 microseconds, and that the task arrival times are dictated by the application access pattern.

To test the effectiveness of our scheduling strategy, we performed experiments with array-based versions of two large, real-life embedded applications: *encr* and *usonic*. *encr* implements an algorithm for digital signature for security. It has two modules, each with eleven processes. The first module generates a cryptographically-secure

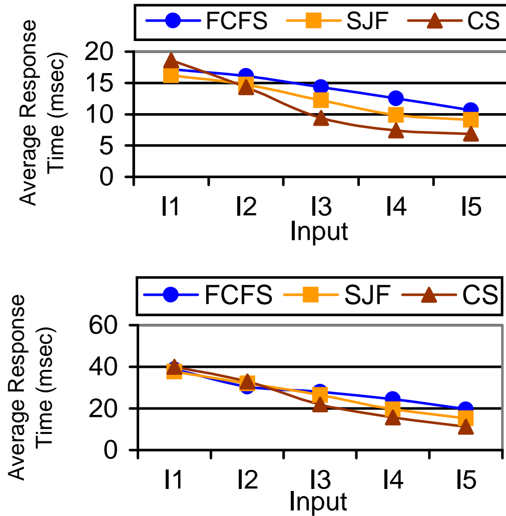

Figure 4. Average response times for three different process scheduling strategies. I1 represents the smallest input size, whereas I5 corresponds to the largest input size. Top: encr; Bottom: usonic.

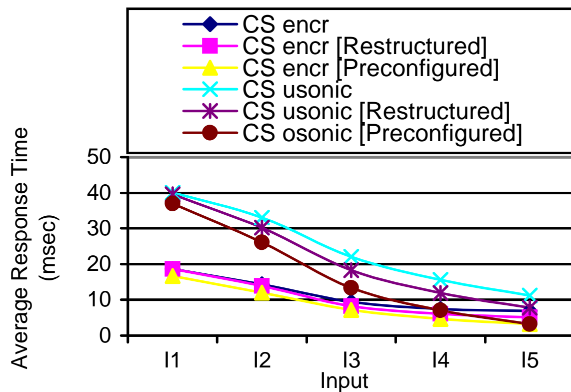

Figure 5. Impact of process code restructuring and preconfiguration on average response times of our scheduling strategy

digital signature for each outgoing packet in a networked architecture, User requests and packet send operations have been implemented as processes. The second module checks the digital signature attached to an incoming message. The main data structure used is an array of lists. The application code contains 355 C lines. Our second application, *usonic*, is a feature-based object estimation algorithm. It stores a set of encoded objects, and given an image and a request, it extracts the potential objects of interest and compares them to the objects stored in the database (which is also updated during the process). It is written in C, consists of twelve processes, and contains 830 lines. For each benchmark code, we performed experiments with five different input sizes (I1 through I5, where I1 is the smallest input size, and I5 is the largest).

6.2. Results

The plots in Figure 4 give the average response time (in msec) for FCFS, SJF, and our scheduling strategy (denoted CS) with different input sizes. In obtaining these curves, we have not employed preconfiguration or process code restructuring. We see that when the input size is very small (I1), the overheads dominate, and there is not much difference between different scheduling algorithms. In other words, the configuration reuse is not very critical. However, with the larger inputs, we observe that the configuration-sensitive (CS) scheduling outperforms the other two algorithms.

Impact of process code restructuring. The curves marked as "Restructured" in Figure 5 illustrate the impact of code restructuring in these two applications. We see that while both the applications benefit from code restructuring, usonic shows larger savings, mainly because there are fewer dependences between the nodes in the STG of this benchmark, and thus, we have more flexibility in re-ordering the processes.

Impact of preconfiguration. The curves marked as "Preconfigured" in Figure 5 illustrate the benefits coming from preconfiguration. It can be observed that, as compared to process code restructuring, this optimization is much more successful. This is because while process code restructuring is not always possible (due to data and control dependences between STG nodes), preconfiguration can be effective most of the time, since the scheduler has advance information as to which process will be executed next.

To better understand the trends illustrated in Figure 5, we also collected statistics that give us the breakdown of the four cases listed in Section 3. Recall that in Case 1, we incur a cost of 0; in Case 2 a cost of reconfiguration; in Case 3 a cost of waiting; and in Case 4 costs of both waiting and reconfiguration. The breakdowns given in Figure 6 clearly show that Case 1 and Case 3 dominate execution behavior, Case 1 taking the biggest share. In other words, our approach makes good use of configuration reuse. The reason that Case 2 does not occur

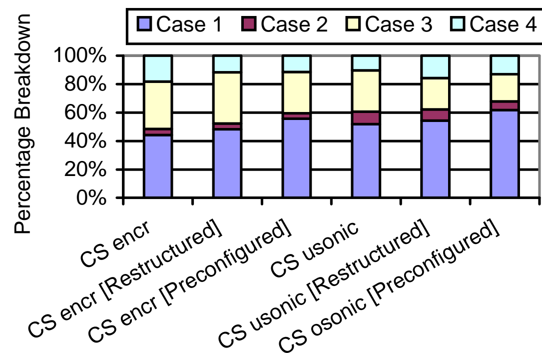

Figure 6. Percentage breakdown of different cases.

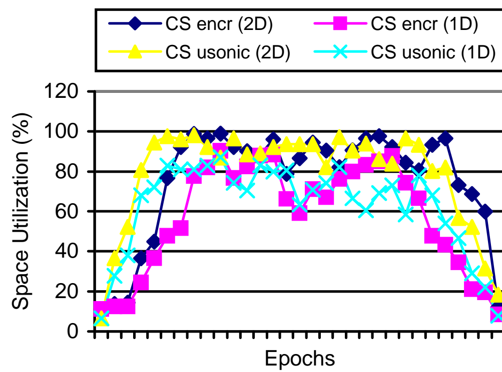

Figure 7. Space utilization over time. 1D = one-dimensional placement; 2D = two-dimensional placement.

very frequently is the fact that it can only occur during the initial phase of execution when not all of the FPGA space has been fully utilized yet.

Another important parameter to study is FPGA space utilization. The graph in Figure 7 shows the space utilization (over the course of execution) for the two benchmarks when no preconfiguration or process code restructuring is used (when they are used, the curves look better) – marked "2D". We observe from these results that the space utilization is quite good, which explains the response time behavior of these applications. It is to be noted that there are two primary reasons, because of which we may not be able to achieve 100% space utilization all the time. First, at the beginning of execution, some FPGA space is not used (until enough STG nodes are scheduled); this is what we observe at the leftmost portion of the curves. Second, since we work with rectangular space allocations, this can lead to fragmentation; that is what we observe beyond the initial portion of the execution. In comparison, the curves marked "1D" give the same results with 1D allocation. That is, each STG node is allocated the entire height of the CLB array, and, as many contiguous columns as are needed to layout the circuit. We observe that while the results with this 1D allocation are not as good as those with the 2D allocation, the former is still able to generate good results. Specifically, Figure 8 presents a comparison of 1D and 2D allocation policies under our configuration-sensitive process scheduling. Comparing this figure with Figure 4, one can see that 1D results are actually competitive with those of SJF and FCFS (even when these two are used in conjunction with 2D placement).

7. Conclusions

The main benefits of FPGA-based computing are the ability to execute larger hardware designs with fewer gates and to realize the flexibility of a software-based solution while retaining the execution speed of a more hardware-centric approach. In this paper, we have

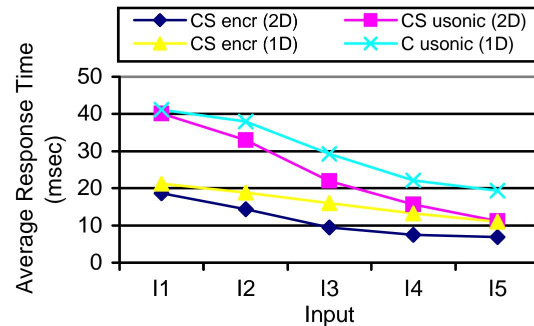

Figure 8. Comparison of 1D and 2D placements.

presented a new OS scheduling algorithm for FPGA-based computation environments. A unique characteristic of this algorithm is that it maximizes configuration reuse, thereby reducing the latency incurred during reconfiguration. In addition, the two optimizations we have proposed, process code restructuring and preconfiguration, help further improve performance over the classical process scheduling strategies.

8. References

[1] M. Barr. A reconfigurable computing primer. In Multimedia Systems Design, September 1998, pp. 44-47.

[2] G. Brebner. A virtual hardware operating system for the Xilinx XC6200. In the 6th International Workshop on Field Programmable Logic and Applications, Springer LNCS 1142, 1996, pp. 327-336.

[3] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. ACM Computing Surveys, Vol. 34, No. 2. pp. 171-210. June 2002.

[4] O. Diessel, D. Kearney, and G. Wigley. A web-based multi-user operating system for reconfigurable computing. In IPPS/SPDP'99, San Juan, Puerto Rico, 1999.

[5] O. Diessel, and H. El Gindy. Runtime compaction of FPGA designs. In 7th International Workshop on Field-Programmable Logic and Applications, Berlin, Germany, 1997.

[6] H. El Gindy, M. Middendorf, H. Schmeck, and B. Schmidt. Task rearrangement on partially reconfigurable FPGAs with restricted buffer. In the 10th International Workshop on Field Programmable Logic and Applications, Springer LNCS 1896, 2000, pp. 379-388.

[7] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor PipeRench: A Reconfigurable Architecture and Compiler. In *IEEE Computer*, Vol.33, No. 4, April 2000.

[8] B. Gunther. SPACE2 as a reconfigurable stream processor. In 4th Australian Conference on Parallel and Real_time Systems, Singapore, 1998.

[9] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 65-74, 1998.

[10] Z. Li, K. Compton, and S. Hauck. Configuration cache management techniques for FPGAs. In IEEE Symposium on FPGAs for Custom Computing Machines, pp. 22-36, 2000.

[11] Virtex-II Family. http://www.xilinx.com /xlnx/xil_prodcat_landingpage.jsp?title=Platform+FPGAs

[12] H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In Design, Automation and Test in Europe, March 3-7, 2003.

[13] G. Wigley and D. Kearney. The development of an operating system for reconfigurable computing. In IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, 2001.