Configuration Scheduling Using Temporal Locality and Kernel Correlation

Santheeban Kandasamy, Andrew Morton, Wayne M. Loucks
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
{s2kandas,arrmorton,wmloucks}@uwaterloo.ca

***Abstract*— This paper examines runtime decisions to configure hardware accelerators or execute in software. Traditionally, reconfigurable FPGAs are reconfigured on-demand with the hardware accelerator, as it is needed by the application. If the software kernel which the hardware kernel replaces is available too, then more sophisticated decision making on reconfigurations may lead to improved execution time and reduced power consumption. The temporal locality algorithm is proposed for applications where individual kernels dominate during differing execution modes. The kernel correlation algorithm is proposed for applications where sequences of kernels are invoked in regular patterns. SystemC simulation is used to compare these two scheduling algorithms against the on-demand policy. Both timing and power consumption results are presented. They indicate that a fairly large reconfiguration time is required for configuration scheduling to be beneficial.**

I. Introduction

This paper examines alternative configuration scheduling policies for run-time reconfigurable FPGAs. It takes the view that reconfiguration is a form of dynamic hardware/software partitioning. Traditionally hardware/software partitioning was performed statically, at compile-time. Through simulation and other techniques, time-critical portions of the application were identified and accelerated with custom hardware. The ability to dynamically reconfigure hardware at runtime, allows the interesting possibility of delaying partitioning until run-time [7], [10]. The advantage of such an approach is that it can react to application behaviour at runtime. The disadvantage is the substantial overhead required for profiling and especially synthesis tools to be available as part of the system. The idea explored in this paper is to use statically configured accelerators but to use smart scheduling of their reconfiguration to adapt to run-time application behaviour.

The architecture being examined is that of a system with a CPU, which will be called a general-purpose processor, GPP, and an FPGA which will be reconfigured with coprocessors, called single-purpose processors, SPP. The FPGAs being considered do not support partial reconfiguration, so they can only hold one SPP at a time. This allows a relatively inexpensive FPGA to be paired with an off-the-shelf CPU. The FPGA is chosen to be big enough to hold the largest SPP that will be used by the application.

Traditional systems use simple on-demand scheduling of the reconfigurable FPGA. (Not to be confused with the "on-demand" reconfiguration of Ullman *et al* [12] for managing partial reconfiguration). The potential problem with simple on-demand scheduling is the time taken to reconfigure the device [3], [4]. The idea here is to explore alternative configuration scheduling algorithms that reduce the frequency of reconfiguration and thereby reduce the time taken in reconfiguration [11]. Reducing the frequency of reconfiguration can also decrease power consumption. Two types of configuration scheduling algorithms are explored in this paper: temporal locality and kernel correlation.

The name "kernel" refers to a part of the application that has been identified has having high execution frequency and/or long execution time. During the design phase, kernels are identified through simulation and profiling and a SPP is created for accelerated execution in hardware. At execution time, the configuration scheduling algorithm decides whether to execute the kernel in software or hardware. It must also schedule a reconfiguration if it is required for a SPP that is not currently configured. The scheduler keeps track of the kernel execution history and uses this information to make scheduling decisions. This is a constrained form of profiling.

In Section II, the two configuration scheduling algorithms are described. Section III describes the simulation of the configuration scheduling algorithms using SystemC and the results. These results include run-time and power consumption.

II. Configuration Scheduling Algorithms

In general, the major phases of a configuration scheduling algorithm can be categorized into the following.

1) Monitoring: Gathering information about the software applications behaviour at runtime.
2) Selection: Deciding which configuration should be configured in the FPGA, using the gathered information from the monitoring phase.
3) Reconfiguration: Performing the actual reconfiguration of the FPGA with the required configurations.

The on-demand configuration scheduling algorithm does not have a monitoring and selection phase, since it simply configures the FPGA with the currently required configuration. The configuration scheduling algorithms considered here have the ability to execute kernels either in software or hardware. Therefore they can be developed such that the monitoring and selection phases are tuned and optimized to the behaviour of the software application at hand.

A. Temporal Locality (TL)

The term "temporal locality" (TL) is borrowed from CPU cache design. It means that if a kernel has been used in the recent past, there is a high probability that it will be used in the near future. Consider the example of a cell phone that also plays music. When it receives a phone call, it goes through the sequence of operational modes of playing music, having a phone conversation and back to playing music. Kernel 1 is used for decoding music files when playing music and kernel 2 is used for decrypting the incoming voice messages of the phone conversation. Kernel 1 only executes when playing music and kernel 2 only executes when the phone conversation is taking place. This software application has the characteristic that different kernels are more dominant and execute more frequently in different operational modes.

A configuration scheduling algorithm can take advantage of this behaviour of the application to accurately predict and schedule configurations of SPP implementations of these kernels. In this study, the temporal locality (TL) configuration scheduling algorithm targets this type of application. In order to illustrate the potential advantage of the temporal locality over the on-demand configuration scheduling algorithm, the following sequence of kernel executions from a software application is used:

$$\text{kernel } 1 \rightarrow \text{kernel } 2 \rightarrow \text{kernel } 1 \rightarrow \text{kernel } 1 \rightarrow \text{kernel } 3 \rightarrow \text{kernel } 1 \rightarrow \text{kernel } 2 \rightarrow \text{kernel } 1$$

It can be seen that kernel 1executes most frequently during the sequence. If it is assumed that the FPGA is initially configured with kernel 1, the on-demand configuration scheduling algorithm would reconfigure the FPGA six times for this sequence of kernels. Each kernel would be executed using their corresponding SPP. A temporal locality configuration scheduling algorithm implementation, on the other hand, could execute the entire sequence without ever reconfiguring the FPGA.

The pseudocode for the temporal locality configuration scheduling algorithm is given in Algorithm 1. A history buffer is used to track recent kernel invocations. The length of the buffer can be tuned to suit the application. The *reconfiguration_required* function simply scans the history buffer to find the kernel with the most invocations in recent history. If the requested kernel has the most invocations but isn't configured, then the function returns **true**. Note that when reconfiguration if required, it is initiated but it does not wait for the reconfiguration to finish. Instead the kernel executes in software. The next time the kernel is invoked the SPP is ready to be used.

Algorithm 1: Temporal Locality (TL) Reconfiguration Scheduling

```
//Monitor Phase
update_history(kernel) ;
if kernel not configured then
    //Select Phase
    if reconfigure_required(kernel) then
        initiate_reconfiguration(kernel) ;
    end
    execute in software ;
else
    start_SPP ;
    wait_for_SPP ;
    get_result_SPP ;
end
```

B. Kernel Correlation (KC)

The term "kernel correlation" (KC) is also borrowed from CPU design. In CPU design, program behaviour at a branch often depends on behaviour at the preceding branch; hence correlated branch prediction. Likewise, the probability of executing a kernel may be correlated with which kernel executed last. Consider the example of a handheld device which is used to watch a streaming video and afterwards is used to listen to streaming music. Both the music and video data is being received by the device over an encrypted wireless channel. The device therefore goes through the operational modes of watching video and then listening to music. Kernel 1 is used for decoding the video file format, kernel 2 is used for decrypting all the incoming data over the wireless channel and kernel 3 is used for decoding the music file format. In the first operational mode of watching video the application decrypts some of the streaming data and then decodes the video format. Therefore kernel 2 is always followed by kernel 1 and vice versa. In the second operational mode of listening to music, the application decrypts some of the streaming data and then decodes the music format. Here kernel 2 is always followed by kernel 3 and vice versa. This application has the characteristic that a critical kernels likelihood to execute in the near future depends on which critical kernel executed last.

The kernel correlation (KC) configuration scheduling algorithm can take advantage of the application behaviour described above. It does this by keeping track of which kernels have executed most frequently after each kernel in the recent past. The pseudocode is listed in Algorithm 2. When a kernel's SPP is not configured, the selection phase decides what SPP should be configured after the current kernel. It initiates the reconfiguration for the next kernel and executes the current kernel in software. If the SPP is configured, then it is used but after use, another selection phase occurs to determine what SPP should be configured next.

One history buffer is maintained for each kernel. It maintains a record of the last n kernels to be invoked following the kernel associated with the buffer. The *reconfiguration_required* function scans the buffer for the current kernel and predicts that the kernel occurring most often in the buffer will follow the current kernel. If that isn't configured, then a reconfiguration is performed.

With temporal locality, the goal was to keep the most frequently executed kernel configured, reducing the reconfiguration frequency. With kernel correlation the goal is to predict the next needed kernel and have it configured before it is invoked.

Algorithm 2: Kernel Correlation (KC) Reconfiguration Scheduling

```
//Monitor Phase
update_history(kernel) ;
if kernel not configured then
    //Select Phase
    if reconfigure_required() then
        initiate_reconfiguration(next_kernel) ;
    end
    execute in software ;
else
    start_SPP ;
    wait_for_SPP ;
    get_result_SPP ;
    //Select Phase
    if reconfigure_required() then
        initiate_reconfiguration(next_kernel) ;
    end
end
```

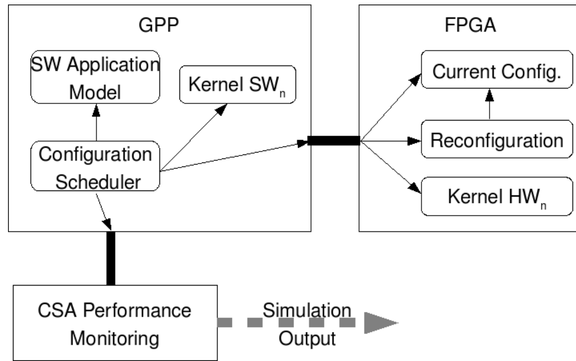


Fig. 1. Simulation Architecture

This might increase the frequency of reconfiguration but since the reconfiguration can be done in parallel to the application software it shouldn't negatively affect execution time. In fact it should reduce execution time since the appropriate SPP will be already configured more of the time, enabling more frequent use of the SPP.

III. SystemC Simulation

A timed functional model was implemented using SystemC to determine the performance of the configuration scheduling algorithms. Timed functional models are used often in hardware/software partitioning since they are useful for analyzing performance trade-offs between different design alternatives [2]. The architecture of the simulation model is shown in Figure 1. The GPP is a 32-bit MIPS running at 100MHz. The FPGA is a Xilinx Virtex-II Pro.

The SystemC simulation does not actually execute the benchmarks: it marks passing of time for each benchmark as it is executed. It also marks the time taken for the configuration scheduling, reconfiguration and executing on the SPP.

To obtain execution times for configuration scheduling, each configuration scheduling algorithm was implemented in C++, compiled and executed on the MIPS SDE instruction set simulator. Execution times were calculated based on the resulting instruction counts, a cycles per instruction (CPI) of 1.5 [13] and clock frequency of 100 MHz.

TABLE I
Test Case Probabilities

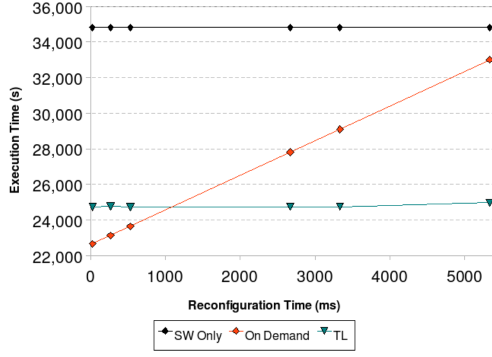
Class	Test Case	Description
Temporal Locality	1	A mode has one dominant kernel ($\sim$ 90% probability), with infrequent execution of other kernels. Modes are repeated 40 times before switching to other modes.
	2	Same as test case 1 but less dominant kernel ($\sim$ 70%).
Kernel Correlation	3	Kernel a is followed by predominantly by kernel b ($\sim$ $85-90\%$ probability) with small probability of other kernels following.
	4	Kernel a is followed less predominantly by kernel b ($\sim$ 70%).

The application execution times were derived from the MediaBench [1] benchmarks: a set of multimedia applications designed for embedded systems [6]. Villarreal *et al* [13] used profiling to identify critical kernels in five of the MediaBench benchmarks. Then they synthesized SPP's to accelerate the kernels. These results were used as the starting point of this case study. The five benchmarks from [13] were combined into one application. Each benchmark has one critical kernel and therefore the entire software application has five kernels, each with a possibility of being executed in software or using a SPP. The timing results from [13] were used as parameters in the simulation.

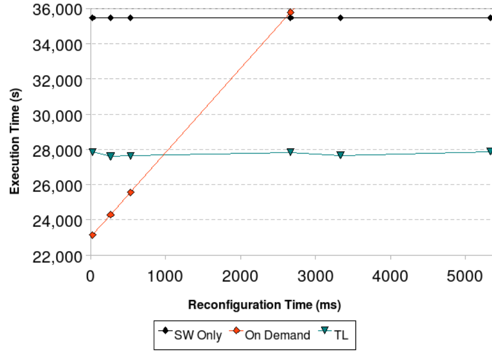
A number of test cases were generated using these five benchmarks as components of an application. The simulation would choose which component to execute next based on probabilities. The probabilities were tuned to represent four types of application as seen in Table I.

The temporal locality scheduling algorithm was used on test cases 1 and 2. The kernel correlation scheduling algorithm was used on test cases 3 and 4. The results are shown in Figures 2 and 3. In each figure, simulated execution time (in seconds) is graphed against reconfiguration time (in milliseconds). The upper line (SW Only) in each graph shows execution time if no SPP is used. The other two lines in each graph compare the configuration scheduling algorithm against simple on-demand scheduling. It can be seen that for small reconfiguration times, on-demand out performs the proposed scheduling algorithms. However as reconfiguration time grows, temporal locality and kernel correlation out perform on-demand. Since the configuration scheduling algorithms sometimes execute the kernel in software (on mispredictions), they are of most benefit when the reconfiguration time is larger than the difference between software and hardware execution time. One would have to analyze the application and relative execution times to choose between on-demand and one of the configuration scheduling algorithms.

[1] http://euler.slu.edu/~fritts/mediabench/

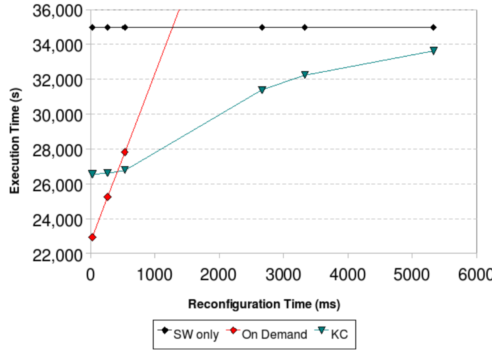


(a) Test Case 1

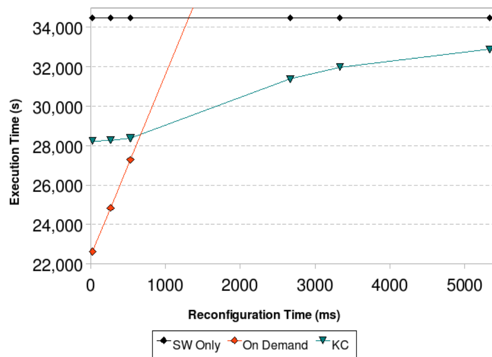


(b) Test Case 2

Fig. 2. Temporal Locality Results



(a) Test Case 3



(b) Test Case 4

Fig. 3. Kernel Correlation Results

TABLE II

Energy and Power Consumption

Test Case	Algorithm	Power	Time	Energy
1	SW Only	105.0 mW	34812 s	3.66 kJ
	On-Demand	104.5 mW	27803 s	2.91 kJ
	TL	104.3 mW	24742 s	2.58 kJ
2	SW Only	105.0 mW	35471 s	3.72 kJ
	On-Demand	105.1 mW	35763 s	3.76 kJ
	TL	104.8 mW	27465 s	2.88 kJ
3	SW Only	105.0 mW	34976 s	3.67 kJ
	On-Demand	104.5 mW	27816 s	2.91 kJ
	KC	104.9 mW	26788 s	2.81 kJ
4	SW Only	105.0 mW	34484 s	3.62 kJ
	On-Demand	104.5 mW	27287 s	2.85 kJ
	KC	105.2 mW	28384 s	2.98 kJ

A. *Energy Consumption Analysis*

Based on the timing results from the simulation, it is possible to estimate the power and energy consumption of the various configuration scheduling algorithms. Power consumption is calculated as:

$$P_{total} = P_{gpp} + P_{FPGA} + P_{ROM}.$$

Power consumption is different when a unit is active or inactive and so depends on the fraction f of time that each unit is active and inactive as stated here:

$$\begin{aligned}
P_{gpp} &= f_{gpp}^{active} P_{gpp}^{active} + (1 - f_{gpp}^{active}) P_{gpp}^{inactive} \\
P_{fpga} &= f_{fpga}^{active} P_{fpga}^{active} + (f_{fpga}^{reconfig}) P_{fpga}^{reconfig} \\
&\quad + (1 - f_{fpga}^{active} - f_{fpga}^{reconfig}) P_{fpga}^{inactive} \\
P_{rom} &= f_{rom}^{active} P_{rom}^{active} + (1 - f_{rom}^{active}) P_{rom}^{inactive}
\end{aligned}$$

P_{gpp}^{active} is the power consumption of the CPU when executing the application in software. At that time the FPGA is inactive and it's power consumption is P_{fpga}^{active}. When execution transfers to the FPGA, roles are reversed. When reconfiguring, the CPU is still considered active and the ROM, which holds the configuration data, is also active at that time.

Power consumption data from [13] was used to estimate the active and inactive power of the GPP. Active and inactive power consumption for the FPGA was estimated using techniques found in [9], [1]. Reconfiguration power consumption was assumed equal to the active power consumption of the FPGA [8]. Active and inactive power consumption for the ROM holding the FPGA configurations was estimated using the technique in [5].

The fraction of time that the GPP, FPGA and ROM are active, inactive or in reconfiguration where obtained from the simulation described above. Based on this data, power and energy consumption is estimated as reported in Table II. For test cases 1 and 2, the reconfiguration time was set at $1000ms$ and for test cases 3 and 4, it was set at $200ms$.

The most important thing to note is the power consumption since the time will depend on the reconfiguration time. The power consumption is affected by the the fraction of time spent

executing on the FPGA and the amount of time spent reconfiguring it. The temporal locality had lower power consumption than the all-software system and the on-demand system. Hence if the execution time is equal then it will have the lowest power consumption. It had lower power consumption because it was the most conservative with reconfigurations, performing them less often than the on-demand system. The kernel correlation had higher power consumption than the all-software system and the on-demand system. It performed reconfigurations more frequently since it was always trying to anticipate the next kernel to be needed. It could still achieve better energy consumption than the on-demand if it's execution time were shorter, as would be the case with longer reconfiguration times.

IV. Conclusions

Two configuration scheduling algorithms have been proposed: temporal locality and kernel correlation. Through simulation using SystemC, it has been shown that the benefit is dependent on reconfiguration time. These simulations used synthetic test cases; real-world applications also need to be obtain more realistic application behaviour. Analysis of power consumption shows that temporal locality reduces power consumption, since it reduced the frequency of reconfiguration. Kernel correlation actually increases power consumption but when that is combined with lower execution times, it can still result in reduced energy consumption.

Which of these alternatives, on-demand, temporal locality and kernel correlation, should be used depends on the runtime behaviour of the application and also the relative speeds of the CPU, FPGA and reconfiguration time.

References

[1] Altera Corporation. Stratix II vs Virtex-4 power comparison & estimation accuracy whitepaper. Technical report, 2005.

[2] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[3] S. Hauck and W. Wilson. Run-length compression techniques for FPGA configurations. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

[4] J. Jacob and P. Chow. Memory interfacing and instruction specification for reconfigurable processors. In *ACM/SIGDA International Symposium on FPGAs*, 1999.

[5] T. Kangas, T. Hmlinen, , and K. Kuusilinna. Scalable architecture for soc video encoders. *Journal of VLSI Signal Processing Systems*, 44(1-2):79–95, 2006.

[6] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, 1997.

[7] R. Lysecky and F. Vahid. A configurable logic architecture for dynamic hardware/software partitioning. In *Design Automation and Test in Europe Conference*, 2004.

[8] L. Shang and N. K. Jha. Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs. In *Conference on Asia South Pacific Design Automation/VLSI Design*, 2002.

[9] L. Shang, A. S. Kaviani, and K. Bathala. Dynamic power consumption in Virtex-II FPGA family. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 157–164, 2002.

[10] G. Stitt, R. Lysecky, and F. Vahid. Dynamic hardware/software partitioning: A first approach. In *Design Automation Conference*, 2003.

[11] S. Swaminathan, R. Tessier, D. Goeckel, and W. Burleson. A dynamically reconfigurable adaptive viterbi decoder. In *ACM/SIGDA International Symposium on FPGAs*, 2002.

[12] M. Ullmann, M. Hübner, B. Grimm, and J. Becker. *On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities*, pages 454–463. Lecture Notes in Computer Science: Field Programmable Logic and Application. Springer Berlin / Heidelberg, 2004.

[13] J. Villarreal, D. Suresh, G. Stitt, F. Vahid, and W. Najjar. Improving software performance with configurable logic. *Kluwer Journal on Design Automation of Embedded Systems*, 7(4):325–339, 2002.