Speeding Up Program Execution Using Reconfigurable Hardware and a Hardware Function Library

Sitanshu Jain  M. Balakrishnan  Anshul Kumar
Shashi Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi 110016
email: mbala@cse.iitd.ernet.in

Abstract
This paper describes a CoDesign environment which follows a new approach for speeding up compute intensive applications. The environment consists of three major components. First, a target architecture consisting of a uniprocessor host and a board with dynamically reconfigurable FPGAs and memory modules; second, a library of functions pre-synthesized for hardware or software implementation; and third, a tool which takes as input an application described in C and partitions it into hardware and software parts at functional granularity using information obtained by profiling the application. An important feature of the partitioning tool is a new efficient heuristic specifically suited for the architecture with reconfigurable hardware.

1 Introduction
Real time applications with strict performance constraints usually calls for CoDesign in order to achieve the right tradeoff between faster and expensive special purpose hardware and cheaper and flexible software. The main task in CoDesign revolves around hardware-software partitioning. A common technique used for partitioning is to extract the computationally expensive portions of the application into hardware, with the remaining executing in software, while meeting all constraints [2]. Alternatively, there are approaches which start from VHDL [3] or Hardware C [1]. Another feature which distinguishes the various CoDesign systems is the partitioning granularity, which varies from fine grain [2] to coarse grain [4, 1]. In the target architecture [1], usually the processor(s) is predicated, whereas the hardware is customized for the application [2].

In our approach, a set of functions are pre-synthesized into hardware designs and organized as a library of hardware functions (called Hardware Function Library or HLIB). This considerably simplifies the overall development cycle. Further, it makes it possible to use exact cost/performance measures during partitioning rather than estimates. This also implies that the partitioning granularity is at function level. We exploit the reconfigurability of the FPGAs to reuse the hardware by dynamically loading different functions at different times during the execution of the application.

In the next section, we give an overview of our complete system, describing the target architecture, the hardware function library, and organization of the software. The algorithm behind this tool is discussed in Section 3. In Section 4, some implementation issues are discussed, and a complete example is shown in Section 5. We conclude in Section 6 with scope for future work on the system.

2 System Overview
2.1 Target Architecture
The target architecture assumed in this paper consists of a single processor with a set of FPGAs interfaced through the system bus [7]. Each FPGA is assumed to have a local memory used for all the data exchanges between the processor and FPGA. This memory is mapped to the processor address space also. A controller, again implemented with an FPGA, coordinates access to the bus, interprets commands for loading FPGAs and monitors status of computations on FPGAs. The architecture assumed is shown in Figure 1. In case a task implemented on an FPGA is to be initiated, the processor fills the FPGA local memory with the data arguments, transfers control to the
FPGA, and waits till it returns. At the time of configuration, the processor copies configuration files to the local memory of the FPGAs and gives a configuration command to load these files onto the FPGAs.

![Diagram](image)

**Figure 1: Target Architecture**

2.2 Hardware Function Library (HLIB)

The Hardware Library is created by identifying functions which are computation intensive, hardware implementable and commonly used in a given application area. Each function is pre-synthesized for a specific FPGA type. For each function, the following information is stored in HLIB:

- Configuration file.
- Compiled code for the processor.
- Performance parameters such as software execution time and hardware execution time.
- Memory address map and size of the parameters.
- Information regarding the function's behavior pertaining to each parameter.

2.3 Software Organization

We use C as the input specification language, with some restrictions on the style of modeling. Structures and user defined types are not supported. Typecasting too is not allowed. The user is also expected to follow a specific modeling scheme, which makes implementation easier, but does not limit the power of the language in any way.

![Diagram](image)

**Figure 2: Design flow**

The design flow of this approach is described in Figure 2 which also indicates the function performed by our tool. In Figure 3, the various modules comprising the tool are depicted.

We have divided the tool into five major modules, namely:

- **Profiler**: generates statistics about the execution behavior of the input application, e.g., the average number of times a loop is executed and the number of times a conditional is successful.
- **Parser**: generates a syntax tree (AST) which represents the input in a dynamic tree structure. It also generates a list of functions used as an input to the partitioner module, and a symbol table.
- **Partitioner**: assigns hardware or software implementation to library functions in the input, based on a heuristics to improve overall performance. HLIB provides quantitative information for partitioning, such as time taken by software and hardware implementations etc.
- **Tree Transformer**: takes the result of partitioning and modifies the AST to ensure the desired implementation of each HLIB function. Manipulations need to be performed to handle data allocation and consistency, since the processor can access the local memory of the FPGAs, but an FPGA has access to only its local memory.
- **Code Generator**: writes the transformed AST
to a C file which is compiled to get the code on the host processor.

3 The Partitioning Algorithm

The algorithm tries to overcome the drawbacks of both exponential time algorithms and greedy ones. It uses sufficient lookahead to avoid local minimas, but iterates only once to obtain fast solutions. Fast algorithm acts as an aid to user in the decision process. The algorithm is divided into two phases: in the first phase, the input is parsed to generate some data structures that aid in lookahead. In the second phase, these are used to decide on the implementation of each HLIF function. The software and hardware execution time is available in HLIF. Also available is the function dependent configuration time for each hardware implementation. Moreover, there is a common configuration time overhead for all functions, independent of data.

Two possibilities arise when we consider partitioning: we allow only static one time configuration, or can allow reconfiguration to be performed during the execution of the application. This choice arises because reconfiguration takes time, and is not useful unless the functions reconfigured are used extensively until the next reconfiguration. Also as we will discuss later, reconfiguration can lead to complications in control dominated applications, and much time may be wasted in order to maintain consistency. On the other hand, static configuration has the drawback of poor utilization of hardware resources.

3.1 Static Configuration

When only static (one time) configuration is used, the overhead of configuration time becomes a one time overhead, and so its offset can be added to all functions in the code. From the statistics obtained from HLIF, and the profile information which gives the total number of calls to each function in the input, it is a simple job to obtain the gain for each function that will be generated if that function is to be mapped onto hardware. From each of these gains, the time to configure that function is subtracted to give an estimate of the effectiveness of the hardware mapping of the function. Now for a target hardware with \( n \) FPGAs, at most \( n \) most effective functions can be picked and configured on the FPGA board at the beginning of the code. Multiple functions implemented on the same FPGA is not allowed since it leads to design contentions in terms of the pins and area available.

3.2 Statically Determined Runtime Reconfiguration (SDRR)

In this approach we allow reconfiguration to be performed at runtime but the decision is taken statically, that is at translation time. Reconfiguration means that the same FPGA may be used for multiple functions during the lifetime of the program, with a configuration file loaded with each new function. Since reconfiguration involves loading of configuration file onto the FPGA local memory, it may not be advisable to reconfigure unless the hardware implementations of the functions involved score greatly over their software implementations. The partitioning problem is therefore two fold - to decide on function implementation, and to decide on the positioning of reconfigurations. We developed a heuristics that does both in polynomial time. Before we describe the heuristics, we define the following:

- **Configuration**: A set of two array variables namely FPGA and CHANGED, which are indexed over number of FPGAs. FPGA[i] gives the function implemented in FPGA #i at that moment. CHANGED[i] is a flag that gets set when the contents of FPGA[i] are either changed or reused.

- **Incoming Weight**: weight attached to a function being processed for implementation determination (in units of time gain that will be produced if this function is implemented on FPGAs.)

\[
W_{in} = W_{cur\_gain} + W_{fat\_gain} + W_{dist} - C_{conf}
\]

where:

- \( W_{cur\_gain} \): gain by implementing the function in
hardware.

$W_{\text{fut-gain}}$ : gain due to later calls to this function.

$W_{\text{dist}}$ : weight corresponding to the distance of next call of the function.

$C_{\text{conf}}$ : Configuration cost.

$C_{\text{conf}} = k \times \text{common configuration time} + \text{data configuration time}$, where $k$ is an empirical constant, $0 \leq k \leq 1$.

- **Inside Weight** : weight attached to a function already implemented on an FPGA.

  $W_{\text{inside}} = W_{\text{fut-gain}} + W_{\text{dist}}$

- **Configuration Weight** : weight of a configuration. This can be defined in two ways based on the following policies:

  **Reconfiguration Avoidance** policy : Reconfiguration is done only when $\text{CHANGED}[i] = 1 \forall i$.

  \[
  W_{\text{config}} = \min\{W_{\text{inside}}(\text{FPGA}[i]) : \text{CHANGED}[i] = 0\};
  \]

  **Reconfigure the weakest** policy : replace the function with the lowest weight, irrespective of whether it is being used in the current configuration or not.

  \[
  W_{\text{config}} = \min\{W_{\text{inside}}(\text{FPGA}[i])\};
  \]

Now the heuristics can be described. There are three different control flow situations which may occur while traversing the list of functions. These are:

i) **A sequence of functions**

In this case we traverse the list of functions sequentially, calculating $W_{\text{in}}$ and $W_{\text{config}}$ using the **Reconfigure the weakest** policy (Figure 4).

ii) **Conditionals**

In this case the strategy depends on whether the

* if path is more frequent or the else path is more frequent or both are equally likely. At the merge of the two paths, the configuration retained is that of the if path, the else path or the intersection of the two configurations respectively.

iii) **Loops**

Inside loops, reconfiguration is not desirable unless time spent inside the loop is large. Therefore we use both SDDR and Static configuration in the loop body, and choose the one with lower cost.

The time complexity of the above described algorithm is $O(nm^2)$, where $n$ is the number of calls to the functions in HLIB, and $m$ is the number of FPGAs. The analysis is omitted here for the sake of brevity.

4 **Some Implementation Issues**

In order to keep the tool portable, it is designed to produce a C output. This output incorporates the decisions generated by the partitioner through transformations at the AST level. These transformations involve additions, deletions and modifications of individual nodes, and are used to make the application execute faster on the template available.

Some major issues addressed in this phase are those of variable allocation, data copying and validity. The local memory of the FPGAs is mapped to the processor memory, and is accessible to it directly. Standard allocation routines are used for variables allocated in processor memory (not in FPGA local memory). Data validity is to be ensured in conditionals, since here the branch taken determines the currently valid location of data. We try to ensure this by keeping home locations where necessary, which keep a valid set of data at the end of conditionals.

Data copying needs to be minimized, because in many applications time taken for a two way data copy can far exceed the gains produced by implementing a function on FPGAs. A copy statement is to be added only when the function being considered is mapped to the FPGA, and the variable is a pointer (otherwise the data is anyhow being copied). Necessary conditions forcing a data segment copy into a location are:

- the data was never present in this location,

- the data was present in the location but it is not valid because its copy at some other location was updated after its last use in this location.

In all other cases, only pointer address need to be changed to the new location.
5 An Illustrative Example

An example showing an input and the library assumed is shown in Figure 5 and the corresponding output generated by the system is depicted in Figure 6. For the sake of brevity, only a small example with a three function library and a two FPGA system is shown. Only the relevant statements are given, while the rest of the code remains unchanged by this tool. The library shows all the information required by the system. In table 1, statistics relating to HLIB functions is given. All values are in μs, and are conservative approximations of real examples used by us.

<table>
<thead>
<tr>
<th>Function</th>
<th>$T_{software}$</th>
<th>$T_{hardware}$</th>
<th>$T_{data}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_x$</td>
<td>100</td>
<td>7</td>
<td>6.25</td>
</tr>
<tr>
<td>$f_y$</td>
<td>159</td>
<td>16</td>
<td>6.25</td>
</tr>
<tr>
<td>$f_z$</td>
<td>36</td>
<td>9.5</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: HLIB statistics

$T_{software}$ and $T_{hardware}$ are the time required by the software and hardware implementations of the function respectively. $T_{data}$ is the time required to setup data arguments for the function. There is also a common configuration time required by the configure operation. This value is estimated to be around 700μs.

It can be seen from the statistics that before the first loop, $f_z$ and $f_y$ are to be configured, and $f_y$ is to be replaced by $f_z$ before the second loop. Although hardware implementation of $f_z$ is much faster than software implementation, reconfiguration is still not performed inside the loop due to the large cost involved. Configuration is handled in the “Configure” function call. Also as was discussed in Section 5, malloc for ‘a’ is removed, and it is assigned the address in FPGA local memory where it is first required. The function “abs_addr” gives the absolute address given an FPGA and a local address in it. All functions mapped to HLIB are linked to a different library (HLIB.h), and the $fpga.no$ is given as an argument for address calculation. Note that since argument ‘a’ is required in two distinct locations, it is copied. “Copy(a, b, n)” copies n bytes starting from

\[1\] Calculated using block transfer rates in the processor to copy configuration files to the local FPGA memory, and time taken to load them.
#include <stdio.h>
#include "SLIB.h"
#include "HLIB.h"

main()
{
    int *a, *d;
    int b, c, count;
    Configure(fx, fy);
    d = abs_addr(2, Zx);
    for (count = 0; count < 10; count++) {
        a = abs_addr(1, Xx);
        HW_fx(1, a, b);
        copy(abs_addr(1, Xx), abs_addr(2, Yy), 100);
        a = abs_addr(2, Yy);
        HW_fy(2, c, a);
        fzd(d);
    }
    Configure(fx, fc);
    a = abs_addr(1, Xx);
    for (count = 0; count < 50; count++) {
        HW_fx(1, a, c);
        HW_fy(2, d);
    }
}

Figure 6: Transformed code generated

location a to location b. As fy does not change 'a', no copy is required when fx is called again. For a, this value is obtained from HLIB.

As is evident from the example, functionality of the application is maintained by the system, since the library files are consistent. In larger applications, multiple reconfigurations are possible, and the effect of partitioning becomes more pronounced.

6 Conclusions and Future work

A complete integrated tool for partitioning of an input specified in C has been developed for an FPGA based CoDesign environment. Currently the user has to specify the system in a restricted language, with some features of the language such as structures, user defined types and typecasting not supported.

The tool uses a novel polynomial time partitioning algorithm. Experimentation with this algorithm has been carried out with approximately real application parameters, and results show that implementing some functions in hardware improves the performance of the system considerably. Further work is planned to allow use of multiple FPGAs for a function and concurrent execution of hardware and software.

7 Acknowledgements

This work was supported by a research grant from DRDO, Min. of Defence, Govt. of India.

References


