CUSTOM INSTRUCTIONS WITH LOCAL MEMORY ELEMENTS WITHOUT EXPENSIVE DMA TRANSFERS

Alok Prakash¹, Christopher T. Clarke² and Thambipillai Srikanthan¹

¹School of Computer Engineering, NTU, Singapore.
Email: {alok0001, astsrikan}@ntu.edu.sg

²Dept. of Electronic and Electrical Engineering, University of Bath, United Kingdom.
Email: eesctc@bath.ac.uk

ABSTRACT

Traditionally, Instruction set extension (ISE) algorithms have treated memory and control flow as invalid operations during custom instruction identification to ensure deterministic latency of these extended instructions. In order to overcome these constraints some work has been done to incorporate local memory for custom instructions with memory operations. Such architectures have invariably relied on the expensive DMA protocol for data transfer. Cache-coherence management poses another challenge in such systems and requires additional hardware and/or software intervention. We propose a novel custom instruction architecture capable of incorporating certain types of memory and control-flow operations. Unlike existing architectures, the proposed design eliminates the need for expensive Direct Memory Access (DMA) transfers and additional cache management sub-systems, thereby saving significant time and energy. Our method is focused mainly on accelerating code segments with static variables as well as the ones allocated on the stack, which are widely prevalent in embedded applications. Experimental results show that the proposed method achieves a substantial performance gain of up to 47% over base processor implementation.

1. INTRODUCTION

Embedded systems are constantly being pushed to achieve higher performance and contend with increasingly strict TTM pressures. In the past decade, FPGA-based designs have slowly gained traction to address such complex requirements. Tremendous research effort has been expended to harness the potential of FPGAs, but a comprehensive design methodology that generates an application-specific system given a particular performance-area constraint, has still not appeared [1]. Instruction set extension is one of the most popular methods to generate an application-specific system especially in FPGA-based designs. In this method, the native instruction set architecture of the processor is augmented with additional custom instructions identified from the application using an offline analysis [2]. The custom instructions thus identified have traditionally considered memory and control as invalid operations in order to have a deterministic latency in the processor’s datapath. A few authors like [3] and [4] showed that including memory and control operations as part of custom instructions can lead to significant performance boost. The authors proposed the use of Local memory alongside the custom hardware to alleviate the memory bandwidth bottleneck [5].

The inclusion of memory operations into hardware blocks necessitates a proper channel for these hardware blocks to access the memory. In previously proposed methods by authors in [6], [7] etc., designers typically transfer the relevant data from the main memory to the local memory of the FPGA (i.e. close to the custom hardware block) by a DMA transfer in a basic block before the custom hardware block executes. After the hardware block executes on the data in its local memory, the data is transferred back to main memory in a suitable basic block using another DMA transfer cycle. It should be noted that a significant portion of time savings achieved by executing the code segment in hardware, is eventually lost due to the DMA transfers. The authors proposed techniques to minimize the effect of these multiple DMA transfers but with limited success. Cache coherence becomes another major issue in such a design and needs to be tackled using additional hardware/software sub-systems [7].

In this work, we propose a novel hardware architecture in which the local memory is connected directly to the main system bus and is in an uncached area of the memory map of the main processor. This model helps us to avoid the expensive DMA transfer costs while also doing away with the cache-coherence problem. We identified and implemented custom instructions with memory and control operations on our architecture. Due to the inclusion of memory and control operations, the selected custom instructions could easily encompass entire basic blocks (a sequential block of code with exactly one entry and one exit point) for hardware execution. By contrast, these operations generally are the non-implementable nodes in traditional custom instructions that do not include memory or control operations. In this work we call the custom instructions with memory and control as Custom Hardware Blocks.
(CHB) keeping in line with basic blocks that can be now be implemented as custom instructions.

We explain and discuss the architecture model in Section 2. We will discuss the identification and selection phases of the CHBs in Section 3. Section 4 details the experimental setup and results.

2. PROPOSED ARCHITECTURE MODEL

The proposed hardware model with Local Memory alongside the custom instruction is shown in Fig. 1. There are two important aspects to be noted in this architecture.

Firstly, as is shown in the figure, we have created a memory mapped slave interface connecting the local memory directly to the main system bus. This ensures that the local memory shares the memory address map with the main memory of the processor. This design allows the processor to access the data from the local memory as it would from the main memory. We constrain the permitted variables stored in the Local Memory to those, which are either static variables or are allocated on the stack in non-re-entrant functions.

Therefore, the proposed model does not require data to be transferred from the main memory to the local memory, thereby avoiding the cost and energy spent in DMA data transfers used in the existing solutions. Instead the relevant variables are located in the Local Memory address space during design time. This is achieved by identifying the arrays of data that are needed by the custom hardware block and storing them in the local memory during design time. When the processor needs to operate on this data set, it can easily access the local memory via the system interconnect fabric.

Secondly, the local memory is implemented in the uncached pages of the system memory hierarchy. This is to ensure that processor always bypasses the cache system while accessing the data from the local memory. This in turn eliminates the need for implementing cache-coherence protocols. Obviously, this leads to longer access times for the data to be accessed from the local memory than the main memory. Therefore, it is vital that we only select relatively isolated code segments for hardware implementation.

We explain this by the following example:

\[ F + H \ll G \]

which means that the rest of the application code rarely accesses the Local Memory LM1. This in turn ensures that the Local Memory LM1 is rarely accessed by the processor (executing the rest of the application not executed by the custom hardware block), but accessed very frequently by the selected custom hardware block CHB1. Therefore even though the access time for the processor to access the data from the uncached local memory is higher than from the cached main memory, there is an overall time saving resulting from this architectural change.

The above example shows the selection of one hardware block. In practice there is more than one hardware block depending on the available area and performance constraints. In such circumstances, the general strategy is to select various code segments accessing a common memory area to be executed as custom hardware blocks simultaneously. The concept is discussed in more detail in Section 3 and then applied to a real-world example from the SHA (Secure Hash Algorithm) application in the MiBench benchmark suite in Section 4.

3. IDENTIFICATION AND SELECTION OF CUSTOM HARDWARE BLOCKS

The proposed implementation process follows the typical two-step process of Identification and Selection to implement the Custom Hardware Blocks. In the Identification phase, we used the Intermediate output (known as Intermediate representation or LLVM-IR) from the open source LLVM compiler infrastructure to identify suitable basic blocks to be implemented as hardware. The LLVM-IR is profiled on the LLVM virtual instruction set to
obtain the execution frequency of various basic blocks. The potential custom hardware blocks were shortlisted from amongst these basic blocks based primarily on their execution frequency. The most frequently executed basic blocks are usually the bottlenecks for software execution and therefore the best candidates for hardware implementation. We used an additional constraint to shortlist basic blocks, with only variables that are either statically allocated or allocated on the stack by a non-re-entrant block, for the Selection phase. Fig. 3 shows a snippet from the sha_transform function in the SHA C Code. In this code snippet, we can easily implement the local variables “A” through “E” and array “W”, since they are copied to the stack before getting used, as part of Local Memory. On the other hand, the structure sha_info cannot be implemented since it is a pointer to a dynamically allocated memory space. The arrays and elements accessed by the memory operations inside the candidate basic blocks in SHA application are listed down in the example of Table 1. It should be noted that although our methodology only works for some variables, it is still useful for embedded

```c
void sha_transform(SHA_INFO *sha_info) {
    int i;
    LONG temp, A, B, C, D, E, W[80];
    ...
}
```

Fig. 3. A code snippet from the sha_transform function in the SHA C code.

Table 1. Most frequently executed basic blocks in the SHA application and corresponding arrays/elements accessed

<table>
<thead>
<tr>
<th>Name of Basic Blocks</th>
<th>%age of total runtime</th>
<th>Variables Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>bb3</td>
<td>38.7</td>
<td>Array “W”</td>
</tr>
<tr>
<td>bb6</td>
<td>12.1</td>
<td>Array “W”, Elem. “A-E”</td>
</tr>
<tr>
<td>bb10</td>
<td>12.1</td>
<td>Array “W”, Elem. “A-E”</td>
</tr>
<tr>
<td>bb13</td>
<td>12.1</td>
<td>Array “W”, Elem. “A-E”</td>
</tr>
<tr>
<td>bb16</td>
<td>12.1</td>
<td>Array “W”, Elem. “A-E”</td>
</tr>
</tbody>
</table>

In this Section, we provide detailed description of our experimental setup. We implemented and verified the proposed hardware model on the Altera DE2 development board containing a Cyclone 2 FPGA device. Altera provides provision for incorporating multi-cycle custom instructions into their NIOS II processor which allows the addition of arbitrary size custom instruction without significantly affecting the processor’s operating frequency.

As explained in the previous section, we propose to select custom hardware blocks based on the relative isolation of their corresponding memory locations. In order to achieve this, we need to identify the arrays accessed by every basic block of interest. We focus only on top most frequently executed basic blocks for this purpose. The LLVM-IR was profiled using the profiler framework in LLVM to obtain a sense of the most frequently basic blocks in the application. Table 1 lists the top most frequently executed basic blocks in the SHA application as obtained from LLVM along with the name of the arrays and elements accessed from these basic blocks. The array “W” in this case has 80 integer elements whereas elements “A” through “E” act as registers for storing 5 intermediate integer values. During pure software execution these arrays and elements are stored in the memory and the processor needs to access them from the main memory through the cache hierarchy. Hence, storing these variables in the local memory alongside the custom hardware block has obvious advantage of quicker access time when the custom hardware block is used for execution.

In the hardware models proposed by [1] - [4] DMA transfers were required to shift these variables from and to the main memory before and after the custom instruction executes. Additionally, there was a possibility of these variables being accessed by the CPU from the Cache while the custom hardware was executing. In order to avoid the usage of wrong data, the authors needed cache-coherence techniques that needed additional HW/SW intervention.

In our design, the array “W” and variables “A-E” were initialized as a pointer to a hardware location in the address space of the local memory. We declared this location to be uncached forcing the CPU to always access this data directly from the local memory instead of the data cache.

In order to verify the hardware model and the proposed concepts, we wrote the Verilog code for the above basic
blocks. These blocks were then incorporated with a NIOS II system as multi-cycle custom instructions. It should be noted that each basic block executed as a custom instruction contained a loop, but still implemented as a custom instruction. In other words, the control operation at the end of the basic block was easily implementable in our architecture. We called these custom instructions from the SHA application C code in a series of experiments with different combinations of these custom instructions. We instantiated a minimally intrusive, “Performance counter peripheral” in the NIOS II system to measure the application runtime in number of clock cycles. The NIOS II system was running at 50MHz clock speed. Fig. 4 shows the total execution runtime of the SHA application with various combinations of the custom instruction. The first column shows the software only execution time without any custom instructions.

As can be seen from the graph, the application runtime generally decreases with addition of more custom hardware blocks. We see a jump in the application runtime from “BB3” to “BB3+BB6” because from Table 1, basic block accesses the elements “A-E” in addition to the array “W”. Again, from the Table 1, we see that the basic blocks “BB10, BB13, BB16” also need to operate on the variables “A-E”. When the basic block “BB6” is executed in hardware, the corresponding variables “A-E” are also stored in the local memory for quicker access. However, the processor needs to access these variables while executing basic blocks “BB10, BB13, BB16”. This is slower than accessing the main memory (via the cache) from the processor and hence the advantage of executing “BB6” as a custom hardware block is offset due to this slow access in “BB10, BB13 and BB16”. In effect, the “BB3+BB6” option increases “G” (due to use of array “W” in BB6), but increases “H” more due to the new penalties from blocks “BB10, BB13 and BB16” accessing elements “A-E” from the uncached memory space. Therefore, it should be noted that it is generally more advantageous for all the basic blocks accessing the same memory variables, to be executed as hardware blocks at the same time. This behavior is also evident from Fig. 4. We see that as we start porting the basic blocks “BB10” through “BB16” into hardware, the application runtime reduces drastically, with the runtime of the last column being about half of the software only execution time.

5. CONCLUSION

In this paper, novel hardware architecture for realizing custom instructions has been proposed that includes memory and control operations. Unlike existing methodologies, our method avoids expensive DMA for data transfer between main memory and local memory. Instead the data was initialized in the local memory during the design phase in uncached pages. This eliminates the need for additional cache-coherence sub-systems. The proposed hardware model and concepts have been verified on a NIOS II processor by instantiating appropriate custom hardware blocks for the SHA application from the MiBench suite. We showed that by implementing the right set of basic blocks in hardware, the SHA application can be run over 47% faster when compared to the pure software implementation.

6. REFERENCES