Modelling Communication Overhead for Accessing Local Memories in Hardware Accelerators

Alok Prakash, Siew-Kei Lam, Thambipillai Srikantan(SM-IEEE)
Center for High Performance Embedded Systems
School of Computer Engineering, NTU
Singapore.
{alok0001, assklam, astsrikan}@ntu.edu.sg

Christopher T. Clarke
Department of EES
University of Bath
United Kingdom
eesctc@bath.ac.uk

Abstract—Local memories increase the efficiency of hardware accelerators by enabling fast accesses to frequently used data. In addition, the access latencies of local memories are deterministic which allows for more accurate evaluation of the system performance during design exploration. We have previously proposed local memories with an un-cached memory slave interface that permits program running on the processor to access the locally stored variables in the hardware accelerator. While this has relaxed the memory constraints for porting code sections to hardware accelerators, there is now a need to consider the read/write access penalties of local memories from the processor during design exploration. In order to facilitate the selection of profitable hardware accelerators, we need an accurate performance model that takes into account these read/write access penalties. In this paper, we propose a novel model to estimate the penalty incurred due to memory dependencies between the program running on the processor and the local memories in the FPGA hardware accelerator. This model can be used in an automated design exploration framework for heterogeneous FPGA platforms to select profitable hardware accelerators with local memories.

Keywords— custom hardware; memory dependency; hardware acceleration.

I. INTRODUCTION

Hardware acceleration of critical code segments in embedded applications has gained wide popularity in the last decade. A large amount of research has been undertaken to make this process seamless and to enable the software programmers to exploit the underlying architecture more efficiently [1]. In order to achieve such design flows, we need tools to analyze the applications and extract various performance metrics to perform efficient hardware-software partitioning. Such tools must also be fast and reliable to allow for a design space exploration before committing the design to hardware. Various authors such as [1] and [2] have proposed high level synthesis tools for mapping C applications on a hybrid architecture consisting of a processor, appropriate hardware accelerators and the memory subsystem. In [3] the authors described a way to compile for such heterogeneous architectures automatically. However, the code sections to be implemented in hardware is selected and decided upon manually and fed into the tool as one of the inputs for design space exploration. This decision does not consider the memory dependencies between the hardware accelerators and the software program. In order to fully automate the tool-chains for selecting suitable hardware accelerators, we need to consider the memory dependencies across various blocks of application code along with other application characteristics that are typically used for design space exploration.

In general, hardware accelerators can be classified into two types based on their implementation. The first type can be considered as micro-accelerators such as custom instructions that are used as additional instructions to the base processor’s instruction set architecture. The term hardware accelerators on the other hand, is used for accelerators composed of larger code segments, which are implemented as a co-processor and hence do not require changes to the processor's instruction set architecture. The term hardware accelerators can lead to hazards. For example consider a situation where a custom instruction has to fetch data from the off chip memory and the processor pipeline has to stall during the fetch cycle, thus incurring a performance penalty. A few authors such as [4] and [5] proposed custom instructions with local memory elements to alleviate this problem. They proposed that the relevant data should be transferred to the local memory elements before the custom instruction executes and then transferred back to the main memory after the execution is completed. Methods to avoid cache-incoherence in such systems have also been proposed.

In our previous work [7] we proposed a heterogeneous architecture that does not need expensive DMA controllers or cache-coherence mechanisms to implement local memory alongside the hardware accelerators. Instead, we proposed to perform an offline analysis of the application to identify high frequency application basic blocks with statically allocated variables. These variables are then placed in the local memory alongside the corresponding basic blocks before the application is executed, thereby obviating the need for DMA transfers. The local memory itself was placed in the un-cached address space of the main memory to avoid the need of cache-coherence protocols.

In this paper we identified the issues that need to be addressed for automating the selection process of such hardware accelerators. Specifically, we list the important metrics that are used during the hardware-software partitioning phase and provide a rapid and reliable way to estimate them. We then derive an equation to estimate the penalty incurred by the software program to access a block of memory instantiated in the FPGA accelerator. This model can be used as part of an automated framework to select profitable hardware accelerators.
that takes into account the memory dependencies between different hardware accelerators and the software program.

The rest of the paper is organized as follows. We describe the application characteristics and the techniques used to estimate them in Section II while Section III explains our penalty estimation model. We prove the effectiveness of the proposed model using a real application in Section IV followed by the conclusion in Section V.

II. CANDIDATE ENUMERATION AND THEIR PERFORMANCE METRICS

As the first step of selecting suitable code blocks for hardware acceleration, one generally performs an enumeration step to list down the possible candidates for implementation. In this step, we use the LLVM [8] compiler infrastructure to do a front end compilation of the application code and obtain a target-independent Intermediate representation (IR). This IR is profiled using the same compiler tool-chain to obtain the execution frequency of all the basic blocks in the application. The most frequently executed code sections or the so called “hot blocks” are the most preferred candidate for hardware acceleration. Therefore we arrange the basic blocks in decreasing order of execution frequency and choose the top most frequently executed basic blocks for further consideration. The number of blocks selected at this stage can be varied according to the user input, but generally only the top 20–30 basic blocks in an application contribute to significant execution frequency.

Once we shortlist these candidate basic blocks, we perform memory analysis to determine which of these basic blocks can potentially be executed in hardware. This decision can be based on certain design constraints which might decree that certain portion of the code must be executed in software. As discussed in our previous work [7], we do not consider basic blocks with dynamically allocated memory for implementation in hardware. Based on such constraints the shortlisted candidates are divided into two categories: Implementable and Un-implementable Basic Blocks.

For the Implementable blocks, we need to obtain various metrics that help us to identify the most suitable hardware candidates for varying performance requirement. These metrics are generally used in any design space exploration algorithm and therefore need to be estimated rapidly for efficient exploration. Here we list down these metrics and provide at least one way to obtain them in a rapid and reliable way.

A. Hardware Area

The amount of FPGA area required to implement a hardware block is an important criterion for selecting a particular hardware block over another. A rapid and reliable estimate of this metric is paramount to any hardware selection algorithm. We have used techniques proposed in our previous work [6] to estimate the FPGA area required to implement the hardware blocks which have been shown to be rapid and reliable.

B. Hardware Execution Time

The time taken to execute a block of code in the FPGA accelerator is the most important criteria for hardware selection. This metric along with the software execution time tells us the amount of time that can be saved by executing a code segment in hardware instead of software. We used the methods described in [6] to estimate the execution time of a basic block in hardware.

C. Frequency of Basic Blocks

The frequency of basic blocks is the dynamic frequency of a block during runtime and is obtained by profiled application code. It is number of times the basic block is executed during run-time.

D. Software Execution Time

The software execution time gives an estimated time required to execute the block of code in software. We estimate the software execution time as the number of basic operations in the block of code. In a realistic scenario, each of the basic operations such as arithmetic, logical etc. take a single clock cycle for execution whereas the memory operations generally need longer time to execute if the relevant data is not found in the cache. With larger available caches in modern FPGA designs this way of estimating the software execution time will not deviate much from the actual execution time.

III. COMMUNICATION OVERHEAD OF USING LOCAL MEMORY BLOCKS

As mentioned earlier, the dependencies of hardware accelerators on multiple memory locations pose an additional challenge to the selection algorithm to choose profitable hardware accelerators. This is due to the fact that there is a penalty to access the variables stored in the un-cached local memories instead of the main memory if the code segment is running in processor. On the other hand a code segment running as a hardware accelerator has almost zero latency for memory access in certain configurations. In order to consider code segments with memory operations for hardware acceleration, we need to account for all the penalties and advantages associated with every memory block in all the candidate code segments.

Figure 1 shows an example hardware model using the Altera’s NIOS II soft core processor as introduced in our earlier work [7]. We will use this figure to explain the various penalties and advantages considered in our selection algorithm.

Figure 1. Example Hardware model using Altera NIOS II.

Consider a block of code “X” being executed in the processor that needs to process an array variable “A”. In a traditional memory hierarchy, the array would be stored in the main memory and will be brought into the processor’s registers through the cache hierarchy. In an ideal situation the array “A”
might already be in the cache and therefore the access latency of this array is minimal.

In a different setup where the system has a hardware accelerator “Y” and the array “A” needs to be processed by the hardware accelerator, it is useful to store the array “A” alongside the hardware accelerator in the local memories. As mentioned earlier, this is useful for two reasons. Firstly, the access latency from the accelerator to the local memory is minimal and secondly this latency is deterministic unlike the access from the main memory. Now, if the code block “X” also needs to access this array “A”, it must access it from the local memory via the system bus instead of the data cache. Obviously this requires a longer access time compared to the access from the cache. The difference in latency of access from “X”, to the local memory instead of the data cache is referred to as the “access penalty”. In order to quantify this access penalty we modeled the time taken for “X” to access the data from both the local memory as well as the data cache. We will define a few terms now, before formalizing an equation to calculate this penalty.

A. Access time from Processor to Main Memory

In order to access a data stored in the main memory the processor needs to communicate through the main system bus, such as the Avalon System Interconnect Fabric in case of Altera NIOS II. The access latency depends on the type of Memory as well as its location. For on-chip memories the access time is obviously faster than off-chip memory modules. Similarly depending on the speed grade of the memory module the access time changes accordingly. We denote the time taken for the processor to access any memory connected via the System bus, but not directly to the processor, as \( T_{PM} \) where,

\[ T_{PM} = \text{Time taken from Processor to Memory.} \]

Since the local memory in our architecture model is tied to the same System Interconnect as the main memory, the time taken by the processor to access the local memory is also \( T_{PM} \).

B. Access time from Processor to Cache

Based on the application requirements most modern processor have a cache hierarchy to store the most frequently used Instruction or Data for faster access. The effectiveness of cache is dependent on several factors such as its size, the loop or data size of the critical code section in the application etc. We denote the time taken for the processor to access the data from its cache memory as \( T_{PC} \) where,

\[ T_{PC} = \text{Time taken from Processor to Cache.} \]

\( T_{PC} \) can be significantly different for different processor architectures and different code sections, varying from the best case scenario of 1 clock cycle to multiple clock cycles in the worst case (due to need of cache flushing etc.).

C. Access time from Hardware accelerates to the Local Memory

The local memory is implemented using the widely available distributed block RAMs in modern FPGAs. The time taken by the accelerator to access any data stored in these Block RAMs is quite deterministic based on the implementation. One can safely assume that the data can be accessed every clock cycle with proper implementation and this time is denoted by \( T_{HM} \) where,

\[ T_{HM} = \text{Time taken from Hardware Accelerator to Local Memory.} \]

Based on the code section for hardware acceleration one can design a multi-ported memory, thereby minimizing \( T_{HM} \) as much as possible through parallel data access.

Now, let us calculate the number of clock cycles saved through implementing a code section “X” in hardware.

\[
\text{Cycles saved} = \left( SW_{Time} - HW_{Time} \right) + \left( T_{PC} - T_{HM} \right) \cdot \text{Freq}_X \]

\[ - \sum_{i=1}^{m} \left( T_{PM} - T_{PC} \right) \cdot \text{Freq}_i \]

\[ ----- \text{Equation (1)} \]

The first term in Equation 1, is the general term used to calculate the number of cycles saved by implementing any software code into hardware. The second term calculates the difference in time of accessing a variable from the cache in case of software implementation and from the local Block RAMS in case of hardware implementation. This difference can be maximized by reducing \( T_{HM} \) as mentioned above. It should also be noted that we have assumed the best case scenario for the software implementation by assuming that the variable is found in the cache. As mentioned earlier, in a more realistic scenario cache performance is dependent on various factors such as the cache size, loop size etc.

The third term is the penalty associated with the processor accessing a block of memory implemented in hardware alongside another code block. This penalty is analogous to the situation where the processor needs to access the data from the main memory instead of the cache hierarchy. It is noteworthy that the summation in (1) considers the penalty incurred when all the basic blocks executed in software, need to access data stored in the FPGA BRAMS. Ideally all the basic blocks accessing the same memory location should be ported to hardware so that there will be zero penalty. However, this is not possible when there is an area constraint for implementing the hardware accelerators. In addition, basic blocks with dynamically allocated memory will not be implemented in hardware. Therefore the main objective of our algorithm is to minimize this penalty as much as possible by selecting the best set of candidates for hardware implementation given the FPGA area and program constraints.

This penalty is also dependent on the latency of the processor for accessing any memory blocks via the system bus which itself can vary with different FPGA devices and families. In our experiments we used the Altera NIOS II processor on Altera DE2 board containing a Cyclone II FPGA. Based on the NIOS II Processor data sheets [9] the latency of the LOAD/STORE operations depend on the choice of the specific NIOS II Core, economy, standard or fast. We used the worst case latency value as stated for the NIOS II economy core i.e. 6 clock cycles for testing the proposed heuristic. The processor to cache latency is stated as one clock cycle per read in the same datasheet. Using these numbers, for a single basic block “i” with “m”
LOAD/STORE operations to a BRAM based memory, the third term can be written as follows:

\[
(T_{PM} - T_{PC}) \cdot \text{Freq}_i
\]

or,

\[
(6 - 1) \cdot m_i \cdot \text{Freq}_i
\]

or,

\[
5 \cdot m_i \cdot \text{Freq}_i
\]

------Equation (2)

Extending equation 2, the total penalty incurred when all the basic blocks in software access the BRAM memory is:

\[
\sum_{i=X-1}^{i=1} 5 \cdot m_i \cdot \text{Freq}_i
\]

------Equation (3)

The upper limit of “i” in Equation 3 shows that the block “X” is assumed to be in hardware and hence we do not need to consider its penalty. We verified this equation, by performing experiments on the Altera DE2 board using different NIOS II configurations (economy/standard/fast). The weight value “5” is definitely dependent on various properties of the FPGA device, the type of processor, the efficiency of its memory management unit etc. We use the current value to prove the effectiveness of the proposed tool and as such its absolute value does not impact the accuracy of the search algorithm. In newer FPGA devices or memory technologies the latency \((T_{PM})\) might reduce, thereby reducing the overall penalty.

IV. EXPERIMENTS AND RESULTS

In order to verify the accuracy of our penalty model, we used a security application SHA from the MiBench [9] benchmark suite as a case study. After identifying the hot blocks in this application, we used the proposed model to estimate the penalty, when various combinations of these blocks are used as hardware accelerators with relevant variables stored in the local memory. To reiterate, the penalty is the communication overhead of accessing these variables stored in the local memory by other code segments currently being executed in the processor. The selection of blocks was done based on the memory dependency across basic blocks and the corresponding penalty. In general a block should not be accelerated until the advantage of hardware acceleration trumps the penalty from the rest of the application currently being executed in the processor.

Figure 2 shows the normalized runtime of the application by accelerating different basic blocks in SHA application. As can be seen from the figure while accelerating BB3 in hardware alone results in about 16% speedup, accelerating BB6 along with BB3 results in reduction in the performance achieved. This can be explained by analyzing the intermediate representation of the SHA application, which shows that BB6 has significant memory dependencies with basic blocks BB10, BB13 and BB16. Therefore if the rest of them are being executed in the processor and access the variables stored in local memory section of BB6, then the overall penalty outweighs the performance gained by hardware acceleration of BB6.

V. CONCLUSION

In this paper we identified the issues that need to be addressed for automating the selection process of hardware accelerators. Specifically, we list the important metrics that are used during the hardware-software partitioning phase and provide a rapid and reliable way to estimate them. We then derive an equation to estimate the penalty incurred when the program executing in the processor accesses a block of memory instantiated in the FPG A accelerator. Experimental results based on a real application clearly show that the proposed model can lead to effective selection of hardware accelerators with local memories.

REFERENCES


