Hardware Libraries: An Architecture for Economic Acceleration in Soft Multi-Core Environments

David Meisner  
Division of Engineering  
Brown University  
Providence, RI 02912  
david.meisner@brown.edu

Sherief Reda  
Division of Engineering  
Brown University  
Providence, RI 02912  
sherief.reda@brown.edu

Abstract

In single processor architectures, computationally-intensive functions are typically accelerated using hardware accelerators, which exploit the concurrency in the function code to achieve a significant speedup over software. The increased design constraints from power density and signal delay have shifted processor architectures in general towards multi-core designs. The migration to multi-core designs introduces the possibility of sharing hardware accelerators between cores. In this paper, we propose the concept of a hardware library, which is a pool of runtime accelerated functions that are accessible by multiple cores. We find that sharing provides significant reductions in the area and logic usage required for hardware acceleration. Contention for these units may exist in certain cases; however, the savings in terms of chip area are more appealing to many applications, particularly the embedded domain. We study the performance implications for our proposal using various multi-core arrangements, with actual implementations in FPGA fabrics. FPGAs are particularly appealing due to their cost effectiveness and the attained area savings enable designers to easily add functionality without significant chip revision. Our results show that is possible to save up to 37% of a chip’s available logic and interconnect resources at a negligible impact (< 3%) to the performance.

1 Introduction

Recently there has been a migration from uni-processor design to multi-core processors. The majority of vendors are quickly turning to this solution in order to solve problems arising from increased spatial integration (e.g. power density, wire delay and complexity). With current technology scaling trends, the number of cores on-chip could be quickly approaching 80 [11]. The introduction of multi-core designs in processor architectures presents new opportunities and challenges to designers. In transitioning to a multi-core processor, simple replication of previous architectures on a die will not suffice for an efficient design [12, 19]. Accordingly, it becomes prudent to reassess the system’s design as a whole, as often one can share a system feature between multiple cores thus freeing up silicon [13, 16]. The reduction in silicon area through sharing is particularly important in Field Programmable Gate Arrays (FPGAs) which are constrained in resources and chip’s area is at a premium price. For example, the largest FPGA from Altera (Stratix II EP2S180C3) cost almost ten thousands dollars as of August 2007. Thus any reduction in silicon area of designs destined for such systems will directly reduce the implementation costs.

Our design’s inspiration comes from the domain of software engineering, where software libraries are created to provide the same functionality to multiple programs without the need for redundant copies of identical code. This task is relatively simple in software because shared libraries can be mapped to the memory space of multiple processes, where the same instructions are read by the various processes with small overhead. Software libraries have the properties of reusability, multiple applications being able to use the same library. It is the later property that inspires this work.

Reusability of hardware enables designers to save both static power and chip area. When not in use, excess hardware leaks current, contributing to power loss and should be eliminated if possible. We propose the concept of a shared hardware library. A shared hardware library exists as a pool of hardware resources or accelerators that any core can use to accelerate the execution of processes running on it. Similar to software libraries, redundancy of resources is avoided by a sharing scheme. Unlike software, however, hardware units cannot be simultaneously used by multiple applica-
There have been recent efforts to optimize the implementation of multiple soft cores in FPGAs. Particularly, one important obstacle is concerned with making the performance of FPGAs comparable to ASICs despite their tendency to consume more area and run at slower clock speeds. Trimming unneeded hardware by Instruction Set Architecture (ISA) subsetting for a given application is one method of reducing power [20]. Another strategy is to reuse hardware that is shared between soft processors. In contrast to our work, previous investigations have focused on small-scale sharing methodologies, such as sharing internal CPU units, e.g. multipliers, between a couple of processors [17]. Another recent suggestion proposes that reconfigurable logic can be mutated by runtime profilers to accelerate and map the binaries executed by a couple of CPUs [13]. This scheme has the overhead of a profiling and mapping system and requires dynamically reconfigurable FPGA fabric.

Many applications require hardware acceleration in order to meet performance requirements for a particular application. For instance, many signal processing applications that require an FFT will not be able to run in real time implemented in software alone. Also, many applications in scientific computing will require evaluation of numerically complex equations or systems. Implementing a specific function in hardware will cut down the runtime drastically while maintaining the general structure and execution flow of general purpose CPU.

The field of biotechnology is driving a demand for increased computational throughput. In particular, DNA sequence alignment and protein identification from mass spectrometry data [5] are becoming computational bottlenecks. The sheer amount of sequencing and mass spectrometry data overwhelms general-purpose processors. Custom hardware acceleration can greatly increase the rate at which the data can be analyzed [15]. Automation of genomic and proteomic sequencing tasks typically uses some form of the edit distance algorithm which efficiently computes the smallest number of changes (e.g., shifts, insertions and deletions) needed to align two strings [14].

The edit distance algorithm relies on dynamic programming, where the overall problem is broken into smaller subproblems and then later combined to give the overall solution [9]. A general framework for the edit-distance function is given in Algorithm 1. The critical section of code is the nested for loops that need to traverse a two-dimensional matrix, presenting the possible alignment scenarios. Given an $n \times n$ matrix, this algorithm would run in $O(n^2)$ time because it has two nested loops of length $n$. With a hardware acceleration, it is possible to exploit the inherent concurrency to run the algorithm in just $O(n)$ time [8, 21]. This gives an impressive speedup over software alone.

Consequently, we use an accelerated version of the edit distance as the backbone of a shared hardware library em-
Algorithm 1 Edit Distance

Input: String A of length N, String B of length M
Output: The edit distance of the two strings

Let \( E \) be a \( M \times N \) matrix

for \( i = 0 \ldots M \) do
    \( E[i,0] = 0 \)
end for

for \( j = 0 \ldots N \) do
    \( E[0,j] = 0 \)
end for

for \( i = 0 \ldots M \) do
    for \( j = 0 \ldots N \) do
        \( E[i,j] = \min( E[i-1,j]+1, E[i,j-1]+1, \text{match}(A[i],B[j])+E[i-1,j-1]) \)
    end for
end for

return \( E[M,N] \)

function \( \text{match}(x, y) \)
if \( x \) equals \( y \) then
    return 0
else
    return 1
end if

bedded in a multi-core environment that is suitable for acceleration of computationally-critical genomic and proteomic tasks. Our proposed concepts apply to any general multi-core system that requires hardware acceleration.

3 System Design with Hardware Libraries

Our proposed system architecture is centered around multiple CPU cores accessing a common shared hardware library. This library contains multiple “functions” that are realized as hardware accelerators. In order to arbitrate requests and data to and from each core, a dispatcher unit exists to mediate. For example, Figure 1 gives the overall organization (showing only one CPU), where a dispatcher unit “sits” between the requesting CPUs and the shared library. A CPU accesses the library by calling custom functions or instructions, just as in software, but in this case, the function calls are trapped and sent to the dispatcher. The dispatcher checks if the library is available to execute the requested functions. In case the library is busy with a previous request, the dispatcher inserts the call into a priority queue, so that the function call is executed once the hardware library is available. Note that the hardware library should be able to access the memory through the system bus. This gives it the ability to receive function calls with pointers in the arguments.

Impact of Sharing on Silicon Real Estate. One of the primary goals of our investigation is to see how the system’s sharing ratio affects the overall system implementation requirements, where the sharing ratio is defined as the ratio between the number of CPUs to the number of accelerators in the system. The main goal for sharing is to reduce the logic needed to synthesize the system. Logic usage is directly related to the real-estate silicon used by the design. Thus decreasing the logic requirement opens the possibility for either adding extra functionality, reducing leakage power dissipation or using a smaller device to house the design. Figure 2 shows a number of possible sharing organizations for a six-core system, where the sharing ratio is equal to 6, 3, 2, and 1. Note that essentially one would like the dispatcher to coordinate calls to multiple accelerators at the same time. Such a scheme will likely improve the system performance, at the expense of a complicated dispatcher design.

The complexity and number of interconnections needed to realize a design is also an important factor when sharing is considered. FGPs have various forms of interconnects that can connect different regions of the chip. It is possible that a design’s logic is complex enough to create congestions that limit routability. Thus, one must consider the interconnect implications of such a sharing mechanism.

Impact of Sharing on Performance. The main performance degradation faced by sharing an accelerator is core stalling, which arises when another core already has a lock
on the library, and another core requests it. Therefore it is informative to analyze the trends in stalling due to different sharing configurations and traffic generation. Two factors directly impact the amount of stalling: (1) the sharing ratio and (2) the rate at which requests are being generated by the cores to the library.

A metric for raw performance of a program on a microprocessor is its execution time. Since the clock frequency remains constant in a design, the number of cycles is directly proportional to the time it takes to execute a program. Another important performance metric of the system is the utilization of the accelerator. The utilization is defined as the total time the accelerator is busy divided by the total time spent with the system running. This can be expressed as follows.

\[ U = \frac{\text{time\_busy}}{\text{time\_total}} = \frac{\#\text{requests} \times \text{time\_request}}{\text{time\_total}} \]

Hence if the hardware library receives many requests over a “small” period of time it will be heavily utilized. Conversely, if in a “long” time span, the accelerator barely gets any requests then it is not heavily utilized. As the sharing ratio increases, there is a more likely chance of stalling, with an increase in library utilization and vice versa. Furthermore, for a given sharing ratio, an increase in the rate of requests from the cores is likely to increase the stalling, with more library utilization. A key question is: given a set of applications, what is the best sharing ratio that leads to the largest savings in silicon real-estate, at a negligible penalty (say <3%) to performance as measured by the execution time? To answer this question, we resort to queueing theory.

**Predicting the Performance Using Queueing Theory.** Queueing theory models the flow of traffic through servers that service customers (for example customers waiting in a checkout line). In computing sciences, queueing theory has been applied to analyzing the scenario of a CPU accessing multiple devices such as hard drive disks [10]. For the proposed sharing systems discussed however, we must model one or more hardware libraries shared by multiple cores. Using queueing theory models, the probability that an individual core \( i \) waits for \( x_i \) cycles before issuing a request for the library is given by the exponential probability density function given below, where \( \lambda_i \) is the average issue rate of core \( i \), or alternatively, the average arrival rate at the accelerator from CPU \( i \).

\[ f(x_i, \lambda_i) = \begin{cases} \lambda_i e^{-\lambda_i x_i} & , x_i \geq 0 \\ 0 & , x \leq 0 \end{cases} \]

Note that the expected value for \( x_i \) is equal to \( 1/\lambda_i \).

Figure 3 plots an example of an exponential distribution for \( 1/\lambda_i = 100 \). Depending on the processes running on the different cores, they might have different average issuing rates. The mixture of a number of exponential distributions with different \( \lambda_i \) leads to an overall gamma distribution. Upon arrival to the dispatcher, each request or function call will be dispatched once the library is available.

**Performance Upper Bound.** We determine a simple upper bound on the impact of sharing on performance. Given a collection of \( m \) cores with each running a process, the worst case scenario occurs when all \( m \) cores request to access a shared hardware library at the same time. In this case, the requests must be serialized. If each core makes a total of \( N \) requests and the library takes \( t \) cycles to complete a request, then the worst case system runtime from the perspective of the last core to be serviced is

\[ \text{Cycles}_{\text{Hardware}} = m \times t \times N. \]

On the other hand, if one were to execute the requests in software instead (with no hardware library acceleration), the running time is determined by a single core. In this case, if the request takes \( k \) cycles to execute in software, the total running time becomes

\[ \text{Cycles}_{\text{Software}} = m \times t \times k. \]
To gain any advantage from acceleration through hardware libraries, \( Cycles_{\text{Hardware}} \) must be less than \( Cycles_{\text{Software}} \), which translates to

\[
m < \frac{k}{\pi}.
\]

Note that \( \frac{k}{\pi} \) is essentially the speedup gained by hardware acceleration. Thus there is essentially no gain in performance if the number of requesting cores are larger than or equal to the speedup attained by the acceleration. For example, if the hardware library offers a \( 6 \times \) speedup over the software, then the number of cores accessing the library should be limited to at most 5 to avoid the worst case scenario. This motivates us to consider hybrid software/hardware library schemes.

**Hybrid Software/Hardware Libraries.** A possible hybrid scheme is to have the shared library in both software and hardware forms. When a core requests a hardware library for acceleration, the dispatcher examines the number of cores waiting for the library. If the number of cores is less than the acceleration speedup offered by the library, the request is queued for later acceleration; otherwise, the dispatcher returns to the calling core with a special return value or through an interrupt mechanism. The return value instructs the calling core to use the software version of the library instead. In principle, the hybrid scheme would have no software engineering overhead because the function would be written in C (or any other language) first before it got compiled to hardware. The only penalty for such a feature would be the increased dispatcher complexity and the software code size.

4 Implementation and Experimental Results

Our system is realized using an Altera Cyclone II FPGA device, manufactured in 90nm technology with 35K logic blocks. We use the NIOS II soft-core RISC processor for our system implementations. The NIOS II processor possesses a custom instruction interface that is taken advantage of to incorporate the dispatcher with the NIOS II cores [7]. For the prototype, all data to and from the main memory goes through the CPU bus. This data will then be fed to the custom instruction and exported to the accelerator as was shown in Figure 1. Each accelerator unit reports to a single dispatcher which can service numerous cores as discussed previously. We have used Altera's SOPC Builder and the Quartus II for implementing the various software and hardware components of the system. As indicated earlier, we have used the edit-distance function as the backbone of our hardware library that is embedded in a system that is designed for accelerating genomic and proteomic computations.

Hardware libraries can be created either through Hardware Definition Languages (HDL), or through software-based languages such as SystemC. The later option makes shared hardware libraries a natural extension to software libraries. In recent years, there has been an increase in commercial tools that directly translate a synthesizable subset of C code to hardware. These include, for example, Altera’s C-to-HDL tool and Celoxica’s Agility compiler [4, 6]. These tools are currently “capitive” to the single core computing paradigm. Our proposed idea of shared hardware library allows current and future tools to migrate to multi-core designs, with graceful demand on the limited FPGA silicon real estate.

We implement our six core system for different sharing configurations as was given in Figure 2. The hardware library takes 16 cycles to finish its computations. We note that our prototype currently allows functional calls using direct operands. The system can be extended to accept pointer arguments, which would be used to access the memory system using Altera’s Avalon bus architecture. Our future implementations can include such access; the only downside would be contention on the data bus that with other cores accessing it as well. We now study the various implications of our system on both the system real estate and performance.

**Logic Usage and Chip Area.** Figure 4 shows the trends in logic usage for the various sharing configurations. Clearly, the higher the sharing ratio, the less logic is needed to obtain the same functionality. Remarkably, the six to one configuration, which has the highest sharing ratio, utilizes nearly one third less of the total logic compared to no sharing at all (the six to six configuration). In fact, the six to six ratio nearly uses all of the Cyclone II device’s available logic.

Figure 5 shows the floor plan of the FPGA for the case of the six-to-three configuration. Each small rectangle represents a CLB region. The darker blue coloring indicates denser logic usage. The magenta represents the area used by the Dispatcher unit(s). Clearly, the Dispatcher and accelerator regions take up a significant amount of space. Increasing the sharing ratio would eliminate up to one of the regions colored in magenta.

**Interconnect Complexity.** The percentage of interconnections used can be seen to increase inversely with the sharing ratio as shown in Figure 6. This trend is logical because the lower ratios require more logic and this increase implies increased connections between this logic. Although, a dispatcher servicing 6 accelerators requires more complicated routing, it remains less than six copies of
Figure 4. Silicon logic usage for the different configurations ratios in a six core system.

Figure 5. Floorplan for the six-to-three configuration.

Figure 6. Interconnect usage for the different sharing configurations.

Figure 7. Average stall time as a function of the sharing ratio and the average issue rate.

Impact on Performance. We run our simulations while varying the parameters of sharing ratio and average request rate $\lambda$. In the first experiment, we assess the impact of varying these two parameters on stalling as given in Figure 7. As would be expected, as the number of cores increases, so does the average stalling. Likewise, an increase in the average rate of request generation also increases stalling. However, the number of stalls does not linearly increase with either of these quantities. One can observe that the increasing number of cores drastically increases the stalling time for any value of $\lambda$ (especially for higher values of $\lambda$). Also, increasing lambda for a given number of cores has a significant negative effect. What these trends infer is continuing to increase the core count or the rate of requests can lead to a major slowdown.

In the second experiment, we assess the impact of sharing ratio and the average issue rate on the execution time in cycles as shown Figure 8. The execution time increases for different issuing rates. However, the magnitude of increase depends on the issuing rate, where small rates tolerate a greater increase in the number of cores.

Figure 9 shows the impact of the request arrival rate ($\lambda$) and the number of cores affecting the utilization of the accelerator. Clearly more cores yield a higher utilization percentage and likewise a higher rate of requests increases utilization. At a certain point, too many cores in the system or too high a request rate will completely saturate the unit and it will always be used throughout the execution. This sce-
### Table 1. Tradeoff between silicon real estate savings for different multi-core environments. Issue Rate (IR) is equal to $\lambda$.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Logic Savings</th>
<th>Interconnect Savings</th>
<th>Performance Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Six to one</td>
<td>33.72%</td>
<td>37.27%</td>
<td>$1.98%$</td>
</tr>
<tr>
<td>Six to two</td>
<td>26.09%</td>
<td>28.77%</td>
<td>$0.03%$</td>
</tr>
<tr>
<td>Six to three</td>
<td>19.53%</td>
<td>21.35%</td>
<td>$0.00%$</td>
</tr>
<tr>
<td>Six to six</td>
<td>0.00%</td>
<td>0.00%</td>
<td>$0.00%$</td>
</tr>
</tbody>
</table>

In the sub-100nm era, designers are no longer able to continue the trend of monotonically increasing clock speeds to improve performance; nevertheless, transistor feature size continues to shrink. In order to utilize the available area, designers incorporate multiple cores on the die. At the same time, FPGAs have risen to prominence for their ability to rapidly enable product development without the soaring cost of custom fabrication. However, designs implemented in FPGAs tend to consume more area and run at slower speeds than designs implemented in ASICs. Perhaps for applications where performance is absolutely the dominant factor, ASICs remain a more viable option. However, when considerations such as power, engineering cost, and device size become important, the proposed sharing method enhances the abilities of FPGAs.

We have proposed the concept of a shared hardware library. This library consists of hardware accelerators that run commonly used algorithms faster than is possible in software alone. Like software libraries these units are shared between multiple instances running programs. However, because hardware cannot be replicated without a cost, one tries to minimize the number of copies required for a given application.

The main goal of a hardware library is to distribute and share multiple hardware accelerators among many cores. By developing a set of guidelines and a general design flow for such a library, engineers can more effectively utilize the available resources in a design to execute their accelerated function. This hardware library would be highly desirable in many systems. In computationally intensive applications, software is often not enough and hardware accelerators are necessary. Moreover, embedded systems in particular have limited chip area yet still require speed in specialized functions. A digital signal processor unit for example, could have separate threads running on multiple cores and then access a FFT unit when necessary. A FFT unit is costly in terms of logic and if multiple processes can share a unit,
there will be a major savings in terms of area with little impact on performance when the system is carefully designed. When area is a precious resource, as in embedded systems, such a setup becomes an attractive option.

The need to minimize the number of copies of the library naturally lends itself to sharing between cores. By creating a dispatching unit, multiple cores can use the same accelerator in turns. Because only one core may use the accelerator at a time, a performance penalty will be incurred if a core needs the accelerator but another core already has the lock. We have shown that one can increase the number of cores sharing an accelerator library only to a certain point after which the performance penalty will become significant. However, with reasonable numbers of core, one gains significant logic and area savings. This will reduce dynamic power consumption and leave more room for additional functionality.

Future revision of this shared hardware library will most likely focus on enhancing the performance of the system. While there are clear advantages in terms of chip economy, performance does degrade slightly for nominal operation. The source of slowdown comes from the cores needing to stall when another core has a lock on the accelerator they wish to use. In this implementation these cores will simply stall until the lock is released.

Acknowledgments. The authors would like to thank Altera Corporation for their generous donation of the FPGA devices and design software. In particular, the authors are grateful to Mike Hutton, Mike Phipps and Steven Brown from Altera’s university support program.

References