Heterogeneity and Reconfigurability as Key Enablers for Energy Efficient Computing

Nithin George
LAP, I&C, EPFL

***Abstract*—**
Driven by performance demands and energy-efficiency requirements, computational systems of the future will most likely be composed of many heterogeneous processing units. In these systems, only a small set of workloads can benefit from a custom functional unit. FPGA-like reconfigurable architectures have the ability to adapt and, thereby, meet the specific computational needs of a wider range of workloads. As current research efforts make these devices more accessible to application developers, we expect these devices to find a place in the computational paradigm in many different domains, integrated in a variety of different ways. Hence, to cover the wide range of possibilities, we need tools that automate the process of selecting, retargeting and executing parts of an application on these reconfigurable devices. Furthermore, there is also a need for domain specific customization of the reconfigurable fabric, which will help to maximize the benefits from reconfigurability.

***Index Terms*—heterogeneous computing, reconfigurable computing, automated customization, reconfigurable architecture**

I. Introduction

THE demand for computational power has constantly risen over the years, driven by applications that are becoming increasingly complex. In the past, however, processor performance kept abreast with this demand through architectural improvements and clock frequency scaling. As advancements in semiconductor technology continue to add more transistors per die, power and thermal dissipation concerns in modern processors have virtually put an end to frequency scaling. To sustain the growth in processor performance, architects resorted to adding multiple processing units on a single die. Additionally, to improve the energy efficiency, modern microprocessors, like the IBM/Sony's Cell Processor [2], Intel's Sandy Bridge [3] and AMD's Llano [4], are embracing heterogeneity. In heterogeneous systems, the individual processing units are specialized to handle specific types of computational tasks. This permits the heterogeneous system to outperform and be more energy-efficient than a homogeneous one. Keeping with this trend, computation systems of the future are likely to add many more heterogeneous units to satisfy the demand for performance and energy-efficiency.

Proposal submitted to committee: September 1st, 2011; Candidacy exam date: September 9th, 2011; Candidacy exam committee: Prof. Babak Falsafi, Prof. Andreas Burg and Prof. Paolo Ienne.

This research plan has been approved:

Date: ______________________

Doctoral candidate: ______________________
(name and signature)

Thesis director: ______________________
(name and signature)

Thesis co-director: ______________________
(if applicable) (name and signature)

Doct. prog. director: ______________________
(R. Urbanke) (signature)

EDIC-ru/05.05.2009

Programming modern heterogeneous systems already represent a significant challenge to developers, and current solutions are often a mix of ad hoc approaches. To make application development for these systems easier, developers need to be assisted by a good programming model. Since heterogeneity in computational systems is expected to grow in the future, a healthy programming model should be able to target a wide range of custom functional units and be easily extended to add new ones. Merge [6] is a promising library based programming model for heterogeneous, multi-core systems. Merge's library based approach enables it to target a wide range of processing units and be extended to support new ones as well. However, the Merge-framework currently does not support reconfigurable platforms.

Reconfigurable platforms, like FPGAs, have shown the ability to adapt to varying workloads and, thereby, improve the energy-efficiency of computation for a wide variety of applications. In the heterogeneous computation paradigm of the future, the ability to reconfigure to match workload pattern would be a welcome feature. However, platforms like FPGAs are yet to gain a significant acceptance beyond certain niche areas. The primary reason for this is, we believe, the difficulty in their programming. When using *Hardware Description Languages* (HDLs), the developer is exposed to the low level hardware details like clock management, state machines, pipelining and explicit memory management etc. One of the research efforts to address this issue is CHiMPS [7], a C-to-HDL compiler that permits developers to retarget C code to FPGAs. In addition to making it easier to write programs for

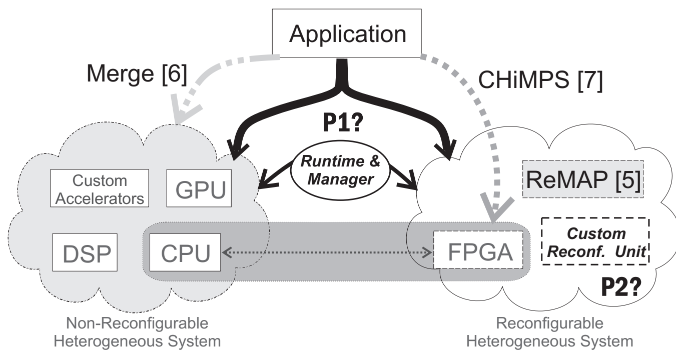

Fig. 1. Considering the computing paradigm of the future, we have Merge [6] that helps developers write programs for heterogeneous hardware, CHiMPS [7] that makes FPGA-CPU systems more accessible, and ReMAP [5] that explores new reconfigurable architectures for future system. However, we still need a system to help programers target reconfigurable, heterogeneous systems of the future. We need a matching Runtime-and-Manager to control the configuration of the reconfigurable fabric and enable them to communicate with the other units in the system. We also need new reconfigurable architectures that better adapted to certain specific domains.

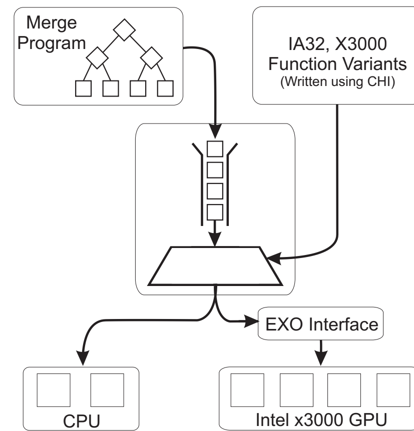

Fig. 2. Sketch of Merge framework. The program, written in the map-reduce pattern, is then compiled into tasks which are pushed onto the task-queue. Work units are dynamically dispatched from the queue, distributing the work among the processing units for which function variants exist in the library. [6]

FPGAs, CHiMPS instantiates multiple caches on the device, which avoids computation stalls due to unavailability of data. Hence, as research efforts mitigate the challenges in programming reconfigurable platforms, like FPGAs, they would increasingly become a part of the heterogeneous computational paradigm.

Figure 1 shows a high-level organization of the reconfigurable, heterogeneous paradigm we expect to see in the future. We believe that depending on the domain and the typical computational workload therein, different systems would opt for different approaches to integrate reconfigurable fabrics. CHiMPS shows one approach, where an FPGA connects to the CPU via the system bus forming a separate accelerator. In ReMAP [5], a reconfigurable architecture designed to accelerate applications on a large-core *Chip Multi-Processor* (CMP), a specially designed reconfigurable fabric is connected to a group of processors as a shared execution unit. Hence, with a slew of different approaches, one topic that interests us, indicated as P1 in Figure 1, is automating the selection, retargeting and execution of parts of an application on the reconfigurable fabric depending to specificities of the system architecture. Another topic of interest, indicated as P2 in Figure 1, is redesigning reconfigurable fabrics to make them better at mapping the workload from a specific domain or for specific purpose; ReMAP is a good proponent of this idea.

The remainder of this document is structured as follows. Section II discusses the selected papers in details and essentially draws on existing research to rationalize the proposed research direction. In Section III, the final section, we propose the direction for future research.

II. Survey of Selected Papers

In this section, we summarize the work done in three papers to gain an idea of what exists and to motivate the direction for future research. The promising ideas we picked for discussion are: Merge–A programming model for heterogeneous systems; CHiMPS–A compiler for programming FPGA-CPU system using C; ReMAP–A novel reconfigurable architecture. Figure 1 portrays the problems tackled by these papers and our focus for future research.

A. Merge: A Programming Model for Multi-Core Heterogeneous Systems

Merge is a general purpose programming model targeting heterogeneous multi-core systems. Merge proposes to replace the ad hoc approaches to programming heterogeneous systems with a library-based methodology that can automatically distribute work among the available processing resources. By dynamically assigning tasks to processing units, deciding the task assignment based on the specific characteristics of the unit, Merge can achieve a high performance and energy-efficiency. Being a library based approach, Merge has the flexibility needed to target systems differing in architecture and processing units.

To target a wide range of heterogeneous processing resources, Merge uses EXOCHI [1] to interface to and write programs for these resources. EXOCHI consists of two parts. *Exoskeleton Sequencer* (EXO) enables the code running on the CPU to interface to the other processing resources. And, *C for Heterogeneous Acceleration* (CHI) permit the programmers to write programs for these accelerators (function intrinsics). Hence, an application written for Merge, when executing, would run function intrinsics written using CHI on the processing resources interfaced to using EXO.

The Merge framework, which can be seen on Figure 2, consists of three components: (1) a high-level parallel programming language which is based on the map-reduce like pattern of functional languages; (2) a predicate-based library system for managing and invoking functions, each of which might have multiple variants targetting different architectures; and (3) a compiler and runtime which together implement the map-reduce pattern by dynamically selecting the best function variant from the library to execute a given task.

Merge applications are written using a parallel programming language that is based on the map-reduce pattern. In this

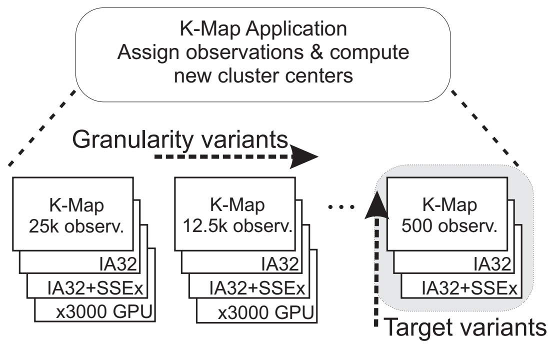

Fig. 3. The hierarchical decomposition of tasks for the k-means algorithm. Granularity variants exist for the given kernel and at each granularity variant, there are multiple target variant that run on different accelerators. [6]

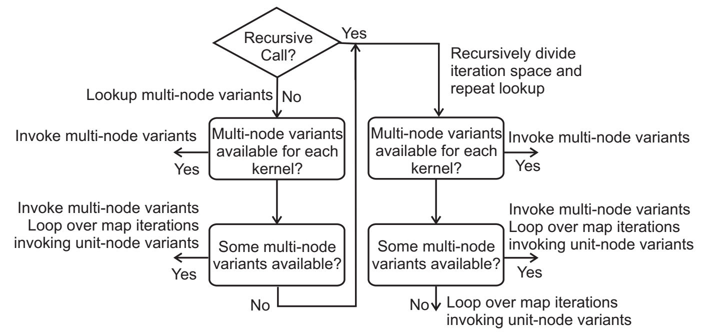

Fig. 4. Runtime variant selection and invocation flowchart. For every task in the task queue, the runtime performs this lookup process to decide on function variants to execute for that given task. [6]

language, all computations are decomposed into a set of map operations and a set of reduce operations. A map operation splits a task into multiple non-overlapping units that can be executed in parallel, and the reduce operation combines the results from these individual map operations to obtain the result for the original task. As shown in Figure 3, a Merge application can use map operations recursively; hence, it can be viewed as being composed of a hierarchical set of tasks that break down the computation into successively smaller operations, called granularity variants. For a given task in that application, the Merge-Library contains multiple implementations, called target variants, which execute on different targets. Hence, at each level in the hierarchy, the application can either target one of the implementations available in the library or choose to decompose that task further into smaller units.

In Merge, the variants for a function that target different accelerators are contained in a predicate-based library. While implementing a function variant, the programmer annotates it by specifying the target architecture, and any other preconditions needed for its correct execution. The annotations in Merge are implemented as predicates or groups. Predicates are logical axioms that include constraints on the structure or size of inputs, and groups are collections of variants formed based on the specific accelerator they run on. These annotations are automatically processed at compile time by the Merge-Library manager, the Bundler, to produce meta-wrappers for the function. Functions performing the same task share the same meta-wrapper. The programmer can now use the meta-wrapper interface to invoke functions. Inside the meta-wrapper, the conditions specified by the annotations are used to decide on a specific function variant to be used for a particular task invocation, thereby, eliminating the need for selective compilation or manual dispatch code. Using these meta-wrappers permits the Merge-Runtime to dynamically distribute work among the available heterogeneous accelerators and also makes it easier to extend an existing system to include new accelerators. The meta-wrappers produced by the bundler also permit querying up target variants and calling a specific variant; this gives the programmer full control over the variant selection process when it is needed.

The Merge-Compiler is responsible for converting the map-reduce statements into standard C++ code, which interfaces with the Merge-Runtime system. First, the compiler translates the map and reduce statements into a multi-dimensional blocked range, each part of which can potentially be executed in parallel. This is possible because, the dataflow of a map-reduce pair can be seen as an implicit tree with the map operation at the leaves and the reduce operation at the joins. Hence, each leaf represents a potentially independent task that can be mapped to a function variant. In the next step, the compiler inserts speculative function lookups to retrieve the applicable variants for each map and reduce operation. The results from the speculative lookup are used by the Merge-Runtime to pick the best variant to use for the operation. Each map or reduce operation, represented by a leaf or join in the tree, is performed by a unit-node function; for multi-core architectures, multi-node function that encompass multiple leaves or joins, performing the equivalent of multiple unit-node functions, can produce better performance. Hence, the Merge-Runtime, besides selecting the best variant, also tries to pick multi-node variants over unit-node ones. This selection process is shown in the Figure 4.

One of the big concerns in Merge is the overhead of the predicate dispatch mechanism and that of the Merge-Runtime. The map-reduce semantics are granularity agnostic and too fine granularity can degrade performance on platforms sensitive to this runtime overhead. Therefore, on such platforms, the decomposition needs to be cut-off at coarser granularities. Single function library synthesis, where unit-node variants are wrapped in loops to create coarser grained function variants, is one technique used in Merge to ensure good performance.

The Merge framework was prototyped on two heterogeneous platforms and tested with a set of informatics benchmarks. The benchmarks were ported into the map-reduce like language and the libraries were enhanced with function variants that would use the different processing unit on the tested system. On a platform consisting of an Intel Core 2 Duo CPU and a 8-core 32-thread Intel Graphics and Media Accelerator X3000, Merge was able to achieve between 3.6 to 8.5 times the performance obtained from running the code on the CPU alone. On a homogeneous 32-way Unisys SMP system with Xeon processors, a performance boost between 5.2 and 22 times was achieved compared to running on a single processor. These results were produced using the same implementation of the benchmarks. Hence, it shows the capability of Merge

to run the same application on differing architectures and use the heterogeneous cores to generate meaningful performance improvement for it.

Discussion: Merge's library based approach permits it to target a wide range of accelerators. The use of the EXOCHI interface in Merge makes it easy to extend it to support new accelerators that support the lightweight EXO interface. This makes Merge a flexible programming model for heterogeneous systems. But, Merge does have some drawbacks. Merge depends on a central control entity to distribute and orchestrate the application execution among the heterogeneous units. Besides, consuming a processing resource, as the number of processing units and the applications running simultaneously on the system increases, this central control can potentially become a bottleneck. In the current implementation, Merge relies on the programmers to decide the hierarchy of function variants. When the number of function variants increases, deciding this hierarchy becomes harder; this is only compounded when the hierarchy of variants change according to size of the data that needs to be processed. Yet another constraint with Merge is the choice of programming language. The map-reduce like programming language requires programmers to explicitly parallelize their applications by decomposing it into a set of map and reduce operations. Chafi et. al [8] use *Domain Specific Languages* (DSLs) with Scala as an embedding language to program heterogeneous hardware. A similar approach can be used to generate code for Merge's underlying framework, making it easier for application developers to use Merge. More importantly, for Merge to support reconfigurable platforms, like FPGAs, an EXO interface needs to be developed for them.

B. CHiMPS: Performance and Power of Cache-Based Reconfigurable Computing

Field Programmable Gate Arrays (FPGAs) are composed of distributed, configurable processing resources and memory resources that are interconnected by a configurable interconnect fabric. This configurability of the FPGA enables it to speed up various applications by either organizing its processing resources into deep pipelines or into customized circuits that exploit the inherent parallelism in the application. The biggest hurdle to the adoption of FGPAs into the general computing paradigm is the difficulty in programming them. Programming in HDLs is very different compared to general programming languages like C. It requires knowledge of hardware concepts like clock-management, state-machines, pipelining etc. Additionally, unlike in general programming languages, developers need to even handle issues like scheduling of data movement and memory access conflict resolution etc.

As stated in the paper, the key to efficiently supporting, C-like, random memory accesses on an FPGAs is to employ caches. CHiMPS, shown in Figure 5, is a C-to-FPGA compiler that mitigates the difficulty of programming commercially available FPGAs and supports caching on them. CHiMPS builds multiple caches using the distributed *Block RAMs* (BRAMs) in the FPGA. Each cache holds data from a particular data structure or memory region, and is customized

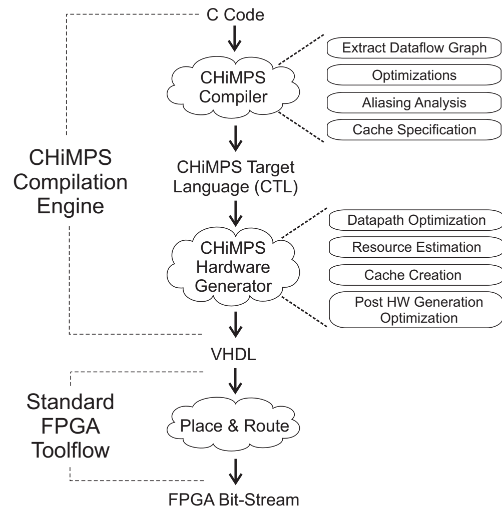

Fig. 5. This is the CHiMPS toolflow. The CHiMPS compilation engine is divided into two main sections: (1) Compiling from C to CTL and (2) Mapping form CTL to VHDL. The VHDL is converted into an FPGA programming file using a commercial CAD tool. Most of the CHiMPS modules have a part in the multi-cache creation: DFG extraction, alias analysis, cache identification and specification, resource estimation, hardware cache generation and post-HW-generation optimization. [7]

to match the characteristics of memory operations on that data structure or in that region. This avoids the bandwidth bottleneck that might arise when using a single, monolithic cache to feed a highly parallel computing fabric in an FPGA. The CHiMPS compilation engine performs analysis and optimizations in two stages that result in (1) adding multiple cache banks, (2) creating multiple caches to increase the memory performance. In the first stage, which is platform agnostic, it compiles the application code into a *Data Flow Graph* (DFG) which is represented using an intermediate language called *Chips Target Language* (CTL). The second stage, which is platform specific, generates the VHDL for the FPGA from the CTL, optimizing the VHDL to get the best performance and optimum resource usage on that specific FPGA. This two-stage approach allows CHiMPS to easily be adapted for different FPGAs. Figure 6 show the outputs generated from the two compilation stages in CHiMPS for given input.

During the first compilation stage, CHiMPS converts all control dependencies into data dependencies to generate the CTL, which is specified in a static single assignment form to increase computation and memory parallelism. To identify opportunities to create multiple, multi-banked caches, CHiMPS identifies the distinct memory regions accessed in the function; it then creates region-specific caches to hold data from these region. A memory region is considered distinct if, during a function call, the pointers pointing to it only point to a location within that region. CHiMPS customizes each region-specific cache according to the memory access pattern and the amount of memory parallelism in that region. Additionally, within each distinct region, CHiMPS identifies potential memory access ordering conflicts between instructions, and mark these dependencies in the CTL. This information ensures that the generated circuit performs the same function as the C code. For nested-loops, during the first phase, CHiMPS identifies the

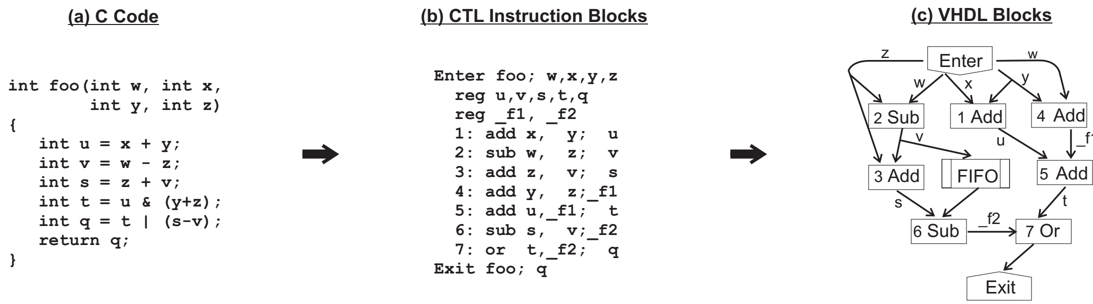

Fig. 6. Outputs from stages of the CHiMPS toolflow. (a) A simple example C function. (b) The C function translated into CHiMPS Target Language (CTL) instruction blocks. (c) The resulting circuit synthesized into the FPGA.

operations that can run in parallel. In cases where the inner loop has tight, loop-carried dependencies, CHiMPS performs loop interchanges to maximize the operation parallelism within the inner loop; this would significantly speed up the loop iteration when synthesized on the FPGA.

Once the DFG is generated in the CTL format, the nodes of the DFG are instantiated as hardware instruction blocks that are connected together according to the graph. The execution scheduler for these hardware blocks is dynamic and distributed, with each block starting execution as soon as its inputs are available and feeding its output to the block it is connects to. In this scheme, multiple instructions can execute in parallel, exploiting both instruction-level parallelism and pipeline parallelism better than on a conventional CPU. To gain the most of this parallelism, the bandwidth of the connection between the computational resources and the caches must be match the parallelism of the memory access operations. To adjust the bandwidth of this connection, CHiMPS analyzes the number of simultaneous reads and writes that can happen to given memory region and decides the number of banks on each cache. However, creating a large, multi-banked cache using a large number of BRAMs would impair the clock frequency at which the cache can operate. Hence, once the number of banks is determined, CHiMPS resizes the cache to ensure the best performance. This is done using a platform-specific model that considers the correlation between the size of the cache, expected miss-rate for that size and the frequency of resulting cache. Additionally, to ensure good performance, while constructing the port arbitration tree to a given cache block, CHiMPS tries to unbalance the tree by giving higher priority to reads and write from the inner loop over accesses from the outer loop. Finally, in the Post-Hardware-Generation phase, CHiMPS tries to improve the performance further by employing two techniques: loop unrolling and tiling. In loop unrolling, CHiMPS replicates the loop instruction blocks and the associated caches to parallelize the computation. The indices of the loop instruction blocks are adjusted so that both the replicas can operate in parallel. In tiling, CHiMPS tries to coarsely partition the data arrays to create separate tiles. Each tiles is now a separate set of hardware blocks and the associated caches that compute on an different array partition. Tiling, hence, improves locality and exploit task-level parallelism to improve performance.

CHiMPS was evaluated on a CPU-FPGA platform consisting of an Intel Xeon processor and an FPGA connected using the processor's *Front Side Bus* (FSB). To guarantee good performance, the FPGA has a low-latency, high-bandwidth access to the main memory with a global, shared address space. The first ensures that large chunks of memory can be accessed with minimum latency; the latter avoids the overhead of copying between different address spaces. For the evaluation, a set of *High Performance Computing* (HPC) kernels were used with sections of code that exhibit good amounts of parallelism being executed on the FPGA and the rest of the code being run on the CPU. For applications where CHiMPS was able to expose the memory parallelism and, thereby, benefit from the many-cache architecture, there was significant speed-up; more than 7 times improvement compared to running on CPU for the benchmarks considered. In experiments where CHiMPS was constrained to generate a single cache or a cache with a single bank, the speed up achieved was much lower, proving the benefit of the many-cache architecture. Hence, parallelism in computation infrastructure will not deliver maximal benefits unless it's supported with a sufficiently high bandwidth memory access, as the many-cache architecture does. The experiments also reveal that parts of the application executed on the FPGA consumed much lower power; on an average, about $1/4^{th}$ of that when run on the CPU.

Discussion: CHiMPS compiler was able to generate from a largely unmodified C code the bit-stream for an FPGA and improve the performance for the evaluated benchmarks. The emphasis in CHiMPS was not maximum performance, but the ease of use. This supports the notion that the real problem with FPGA like reconfigurable logic is their accessibility rather than performance benefits. However, CHiMPS uses an FPGA as a separate accelerator to speed-up whole functions. Hence, it does not cover the case of using the FPGA as a functional unit for fine-grained co-operation with the CPU. Additionally, in a full system, to reap the benefit of reconfigurability, there need to be an additional system that manages the configuration of the FPGA, changing it depending on the workload's computational requirements.

C. *ReMAP: A Reconfigurable Heterogeneous Multicore Architecture*

ReMAP is a reconfigurable architecture that is geared towards accelerating and parallelizing applications within a large-scale, heterogeneous CMP. In a CMP, the reconfigurable

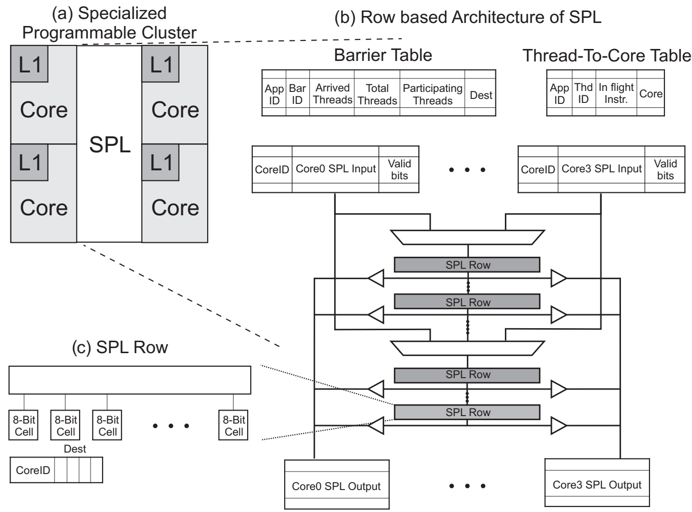

Fig. 7. ReMAP heterogeneous CMP. (a) The SPL cluster with four separate cores attached to the reconfigurable fabric; (b) Detailed view of the row based architecture of the four-way shared SPL cluster including the tables required to support inter-thread communication, and (c) Design of a single SPL row. [5]

fabric can be used to accelerate both the computation in single-threaded applications and the computation and communication in multi-threaded applications.

ReMAP consists of clusters of generic processing cores and reconfigurable clusters, shown in Figure 7, composed of processing cores that share a reconfigurable fabric. The reconfigurable fabric in ReMAP is called *Specialized Programmable Logic (SPL)* and it is shared temporally in a round-robin fashion among the four cores in the same cluster. The SPL interfaces to the cores in the cluster as a reconfigurable functional unit and it connects to the memory system using a decoupled, queue-based architecture. To reduce contention among the cores due to sharing, the SPL can also be partitioned among the cores where each partition is used, independent of one another, for different types of computation.

As shown in Figure 7, the SPL is a highly pipelined structure that is composed of 24 rows. Each row contains 16 cells and each cell is responsible for computing on 8-bits of data. A cell performs the same computation on all the 8-bits it receives. To perform this computation, each cell contains a 4-input LUT, some 2-input LUTs, a fast carry-chain, barrel shifters and flip-flops. This design of the SPL enables it to accelerate single-threaded applications. To support multi-threaded applications, SPL contain additional structures like the *Thread-to-Core Table* and the *Barrier Table*. The *Thread-to-Core Table* enables fine-grained communication between the cores that share the SPL; the *Barrier Table* assists the threads running on these cores to synchronize at a barrier.

The row-based structure of SPL makes it easy to express the computation requirement of any operation in terms of the number of SPL rows that are needed. If an operation needs more rows than is available to the core running that operation, the function can be virtualized on the fabric. When a function is virtualized, a single physical row is used to perform the computation of multiple virtual rows at a cost of performance. This ability to virtualize an operation means that operations needing the SPL can continue to execute even when the number of free SPL rows available is lesser than the operation's demand.

There are three main ways ReMAP's architecture enables it to accelerate computation. These are 1) using the SPL for computation, 2) using the SPL for fine-grained inter-thread communication, and 3) using the SPL for barrier synchronization. Additionally, in the case of inter-thread communication, the SPL can also be configured to perform some computation on the data being communicated between the cores; and, in the case of barrier synchronization, the SPL can be used to perform a global function when the barrier is hit and to communicate the computed result to all the threads participating in the barrier.

For computation workloads that are suitable to be executed on the SPL, the structure of the SPL permits multiple operations to remain in flight and, hence, produce a higher effective throughput compared to executing it on the processor core. Consider the computation shown the flowchart in Figure 8(a); if the computation of *Block 2* is a good candidate to be accelerated by the SPL, it can be performed in the SPL. This has been shown on Figure 8(b). The benefit of using the SPL in this manner, to accelerate single-threaded execution, was also seen in the experimental results where performance improvements of up to 4 times were obtained in certain cases.

ReMAP's architecture also enables efficient fine-grained communication among the threads sharing the fabric, creating parallelization opportunities in multi-threaded applications that are too costly for conventional software-based methods. The *Thread-to-Core Table* in the SPL permits the output of an operation started by one core to be sent to the output queue of another core, hence, enabling fine-grained communication between threads. Such fine-grained communication greatly benefits applications having a producer-consumer type of interaction between threads. In a classic producer-consumer application, where the SPL acts as the communication queue between the producer and the consumer, each thread can run independently as long as the queue is not empty/full. Additionally, some of the computation from either the producer or consumer can be pushed onto the SPL; here, the SPL performs computation as the data moves from the queue-input of one core to the queue-output of another. This ability of the SPL to enable efficient, fine-grained communication, along with computation, can benefit highly pipelined and streaming applications.

For the computation on the flow-chart in Figure 8(a), Figure 8(c) illustrates how SPL can be used to accelerate it using two threads working in a producer-consumer relationship. Here, the computation of *Block 1* is performed on one core and that of *Block 2* and *Block 3* are performed on another core. In this case, the SPL is used as queue to transfer data between the two threads. In Figure 8(d), the computation of *Block 2* is now moved into the SPL where it is performed as the data moves from the producer to the consumer thread.

The ability to perform some computation on the SPL is extremely useful because it can help to balance the work done at the producer and at the consumer. This alleviates stalls

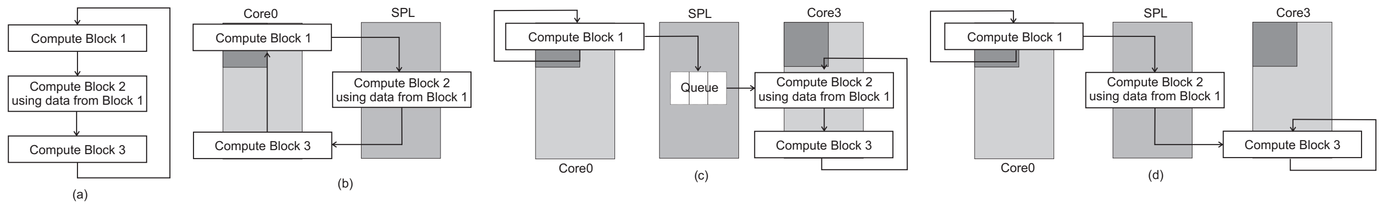

Fig. 8. Computation workload and Communication workload. (a) The flowchart of the processing task performed using the SPL, (b) Performing the computation using the SPL only for computation, (c) Performing the computation using two cores of the SPL cluster in a producer-consumer configuration and using the SPL as a queue, (d) Performing the computation in using two cores of the SPL cluster in a producer-consumer configuration and using the SPL to perform computation on the data as it moves form the producer to the consumer. [5]

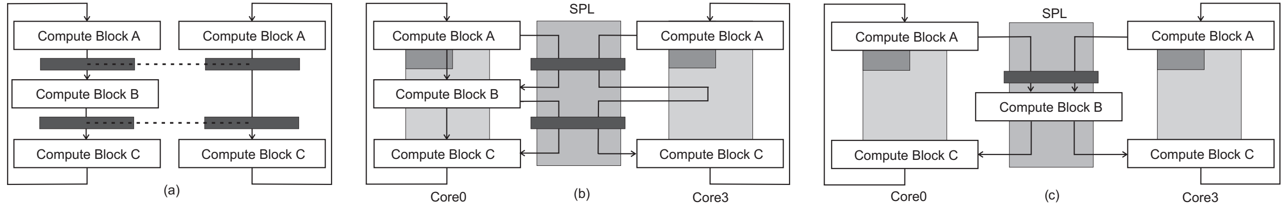

Fig. 9. Barrier synchronization and barrier synchronization along with computation. (a) The flowchart of the task needed synchronization at a barrier, (b) Performing the barrier synchronization alone using SPL (b) Performing barrier synchronization followed by some computation on the SPL. In this scheme, the need for the second barrier is removed as the cores wait until the SPL finishes the computation task. [5]

due to the difference in the rate of data production and the rate of data consumption, permitting more efficient pipelining. Moving conditional statements into the SPL can reduce the stalls due to branch misprediction. Performing computation in the SPL decreases the number of instructions executed at either the producer or the consumer, which improves their execution times and also reduces the pressure on structures like ROB, instruction cache etc. All these factors contribute to make performing fine-grained communication on the SPL very beneficial, as also seen from the experimental results. The experimental results also reveal that using the SPL to perform both communication and computation results in the best performance and energy-efficiency.

Barriers are one of the most common means to synchronize threads in multithreaded applications. The overhead of performing barrier synchronization can be quite significant, particularly when the thread count is high. The *Barrier Table* in SPL enables it to synchronize among the threads executed on cores attached to it. To perform barrier synchronization, the SPL has a barrier instruction that remains blocked until all the participating threads arrive at the barrier. Additionally, in cases where a barrier is followed by a serial function performed on one of the threads, this function can be synthesized on the SPL with the computation output being communicated to all the participant cores. In this manner, the SPL can also improve upon dedicated schemes that support barrier synchronization alone.

In the execution flow shown in Figure 9(a), the two threads need to synchronize at two barrier. The SPL can be used to perform the barrier synchronization, as shown in Figure 9(b). However, the best performance is obtained when the SPL is used to perform both the barrier synchronization, as well as, the computation in *Block B*, as shown in Figure 9(c). In this case, the need for the second barrier is also averted as the threads remain blocked until the SPL communicates the result of the computation. The experimental results reveal that using the SPL to perform barriers synchronizations improved the performance of selected benchmarks. However, the best performance and energy-efficiency is obtained while performing both the barrier synchronization and the computation.

Discussion: ReMAP manages to improve the performance and energy-efficiency for both single-threaded and multi-threaded benchmarks using a specifically designed reconfigurable fabric. The row-based reconfigurable fabric, which is different in comparison to that of an FPGA, exemplifies the benefits of redesigning the reconfigurable fabric to suit different processing needs. However, currently, ReMAP has no compiler support and the benchmarks used for the test had to be manually partitioned to be run on the SPL cluster and normal cluster. Hence, it is yet unclear if a compiler can be developed that will exploit the full potential of a highly flexible architecture like ReMAP. Without compiler support, the benefits of ReMAP would remain, largely, inaccessible to developers of applications.

III. Future Research Direction

As noted in ReMAP [5], reconfiguration platforms can provide high performance and energy efficient computation on workloads that maps well onto them. This makes them very interesting computing platforms for data centers and mobile devices where power and energy-efficiency are major concerns. However, in spite of their suitability, FPGA like reconfigurable platforms have not gained much acceptance, except for certain niche areas like high performance computing or some ad hoc application specific computing.

We believe that one of the possible reasons for this trend is the lack of suitable programming models that cover platforms like FPGAs, thereby, making it harder for application developers to use them. Merge [6] provides a library based programming model for heterogeneous systems that, we believe,

can be extended to enable the integration of reconfigurable units, like FPGAs. The other, perhaps more serious, problem that limits the visibility of FPGAs as a general computation platform is the incompatibility of high-level languages in von Neumann model to program it. CHiMPS [7], the C to FPGA compiler, was able to generate from a largely unmodified C code FPGA bit-stream that produced a significant performance improvement, compared to running the C code on the CPU. With a lot of the difficulties in programming being mitigated and due to the inherent energy efficiency of FPGAs over CPUs, it seems inevitable that reconfigurable fabrics, like FPGAs, would gain popularity in the heterogeneous computational paradigm. Hence, in general, our objective is to work on problems relating to adding reconfigurability into the heterogeneous computing infrastructure of the future.

The problem we see is that integrating reconfigurability into the computing paradigm, however, will most likely be domain dependent. In domains like high-performance computing or cloud computing, the volume of data being computed upon is typically quite large. Hence, the computation kernels that can be run on the reconfigurable fabrics might be working on a huge amount of data, perhaps, working in parallel to and independent of the CPU for a significant amount of time. This suggests that adding a reconfigurable element like an FPGA as a separate accelerator might be a good approach in these domains. However, in other applications like mobile computing, where the amount of data processed is small, the objective might be to assist the main processor and, thereby, increase the performance and energy-efficiency of computation. Hence, in the mobile domain a good approach might be to integrate the reconfigurability into the main processor, permitting fine-grained task assignment and data sharing between the processor and the reconfigurable fabric.

The question of how and where to integrate the reconfigurable elements can also depend on the typical workload handled by the system. A good example to elucidate this is a project we are working on where we attempt to accelerate databases that use solid-state disks as the back-end storage media. Database queries like table scans have a huge amount of parallelism, and for data-analytics they form a significant portion of the system workload. For these queries, the data access is the bottleneck, with the limited bandwidth on the interface to the back-end storage being the main culprit. In a solid-state disk, the data is stored on multiple flash chips and hence can be accessed in parallel. Therefore, in order to accelerate these database queries, we integrated an FPGA close to the flash media, connecting it directly to the flash controllers. While executing a query, the host-processor configures the FPGA resources to perform some of the computation close to the data and, thereby, reducing the amount of data that needs to be sent over the interface. Based on the objective to accelerate table scans, the operations mapped on the FPGA were projection, some filtering and some aggregation operations performed on the database pages. This choice of operations was based on their suitability to be mapped efficiently on the FPGA and the need to reduce the total data communicated over the interface.

Hence, considering different scenarios, there are a plethora of different schemes for integrating a reconfigurable fabric into a system. Depending on the specificities of the domain, the computational workload and how the reconfigurable fabric has been integrated, the kind of computation that can be beneficially accelerated using the reconfigurable fabric varies. Hence, to cover the wide the range of possibilities, we need to develop tools that will help us automatically identify and retarget parts of the application to be executed on the reconfigurable fabric. In addition to this, we also need to design runtime systems that will manage the configuration of the reconfigurable resource and schedule tasks to be run on it. This is one topic we would like to look into during my research.

Another topic that interests us is designing reconfigurable fabrics that can efficiently map the workloads from a given domain, like mobile computing, better than the commercial FPGAs available today. The problem we see is that the design focus of FPGAs has mostly been flexibility. A good indicator of this is that reconfigurable tile in the FPGA is designed for computation on bits rather than on bytes or words as processors cores are. This suggests that there might be a scope to modify the FPGA's reconfigurable tile to make it more suitable to map computation task on. ReMAP's [5] reconfigurable tile, which looks very different to an FPGA tile, supports this argument. In markets where the computation workload can vary significantly and energy-efficiency is key, like in the mobile computation space, reconfigurable fabrics tailored to specific needs on the domain can be better to help reap the benefits of reconfigurability.

References

[1] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G. Lueh, and H. Wang. 2007. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation* (PLDI '07). Pages 156-166.

[2] The Cell Project at IBM Research http://researchweb.watson.ibm.com/cell/

[3] O. Granatir. A Look at Sandy Bridge: Integrating Graphics into the CPU http://software.intel.com/en-us/blogs/2011/01/13/a-look-at-sandy-bridge-integrating-graphics-into-the-cpu/?wapkw=(sandy+bridge+microarchitecture)

[4] D. Scansen. Closer look inside AMD's Llano APU at ISSCC http://www.eetimes.com/electronics-news/4087527/Closer-look-inside-AMD-s-Llano-APU-at-ISSCC

[5] M. A. Watkins and D. H. Albonesi. 2010. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '43). Pages 497-508.

[6] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. 2008. Merge: A Programming Model for Heterogeneous Multi-Core Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XIII). Pages 287-296.

[7] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig. 2009. Performance and Power of Cache-based Reconfigurable Computing. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (ISCA '09). Pages 395-405.

[8] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. 2011. A Domain-Specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (PPoPP '11). Pages 35-46.