Rethinking FPGA Computing with a Many-Core Approach

John Wawrzynek, Mingjie Lin, Ilia Lebedev, Shaoyi Cheng, Daniel Burke

Department of Electrical Engineering and Computer Science, University of California, Berkeley

***Abstract*— While ASIC design and manufacturing costs are soaring with each new technology node, the computing power and logic capacity of modern FPGAs steadily advances. Therefore, high-performance computing with FPGA-based system becomes increasingly attractive and viable. Unfortunately, truly unleashing the computing potential of FPGAs often stipulates cumbersome HDL programming and laborious manual optimization. To circumvent such challenges, we propose a *Many-core Approach to Reconfigurable Computing* (MARC) that (i) allows programmers to easily express parallelism through a high-level programming language, (ii) supports coarse-grain multithreading and dataflow-style fine-grain threading while permitting bit-level resource control, and (iii) greatly reduces the effort required to repurpose the hardware system for different algorithms or different applications.**

Leveraging a many-core architectural template, sophisticated logic synthesizing techniques, and state-of-art compiler optimization technology, a MARC system enables efficient high-performance computing for applications expressed with imperative programming languages such as C/C++ by exploiting abundant special FPGA resources such as distributed block memories and DSP blocks to implement complete single-chip high efficiency many-core microarchitectures.

To quantitatively validate the proposed MARC system, we implemented a MARC prototype machine consisting of one control processing core and 32 arithmetic processing cores using a Virtex-5 (XCV5LX155T-2) FPGA. For a well-known general-purpose Bayesian computing problem, we compare the throughput and runtime of this MARC machine, with fully synthesized application-specific processing cores, against a manually optimized FPGA implementation—BCM (Bayesian Computing Machine) [1]. As the problem sizes range from 10^3 to 10^6, this MARC machine achieve 8.13 GFLOPS in throughput on average, which is 43% of that of BCM but with much less design/implementation effort and much greater portability and retargetability. More importantly, we developed a simple analytical performance model to explain the performance discrepancy between the MARC machine and the hand-optimized BCM FPGA implementation [1].

I. Introduction

FPGA computing can achieve hardware efficiency much closer to ASIC solutions than many alternative processor-based computing systems. Specifically, the computing elements within modern FPGAs, due to their implementation flexibility, possess huge potential to extract application-specific parallelism, therefore often enabling power-efficient computation in ways not possible on traditional multiprocessor systems. Consequently, an FPGA-based computing system frequently results in orders-of-magnitude higher overall performance than that of CPU- or even GPU-based platforms [2], [1]. Today, the advantages of FPGA-based computing become increasingly pronounced because ASIC design and manufacturing costs have sky-rocketed with each new technology node. As a result, embedded and high-performance computing research faces new opportunities to accelerate energy-efficient computation with FPGA-based programmable hardware platforms, but *not* without significant challenges.

One significant barrier preventing a wider adoption of FPGA computing is its lack of a coherent computational abstraction. As a result, fully realizing FPGA's performance and efficiency potential often requires cumbersome HDL programming and laborious manual optimizations. Specifically, programming FPGAs demands skills and techniques well outside the application-oriented expertise of many developers, thus forcing them to step beyond their traditional programming abstractions and embrace hardware design concepts, such as clock management, state machines, pipelining, and device-specific memory management. Furthermore, because memory access is the main performance bottleneck in many applications, complex ad hoc memory architectures are often designed and used, complicating the reuse and maintenance of the resulted implementation. Therefore, it is imperative to create new design methodologies to reduce the cost of generating reconfigurable/customized hardware solutions for FPGAs, making them a more practical computing platform.

A. Landscape of FPGA Computing

Differing greatly in computing efficiency, hardware flexibility, and ease of use, various platforms exist to accomplish computing tasks. From general-purpose processors to ASICs, as the hardware flexibility of these computing devices decreases, their computing performance or efficiency improves significantly (Fig. 1). Similarly, constraining the usage model of reconfigurable devices will also likely result in performance degradation compared with fully customizable solutions. Interestingly, there is a sizable design space (the gray area in Fig. 1), between hand optimized FPGA solutions and general-purpose processors, that warrants being systematically explored.

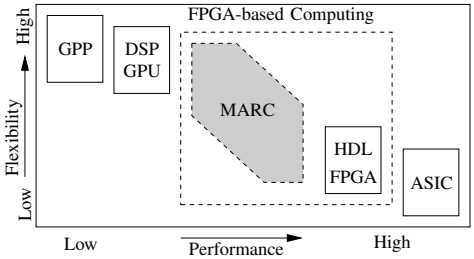


Fig. 1. Landscape of modern computing: flexibility vs. performance.

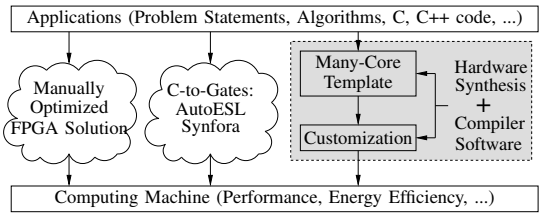


Fig. 2. Three methodologies to accomplish FPGA computing.

Within FPGA-based computing (Fig. 2), there are three promising approaches: manually optimized application-specific solutions, C-to-gates approaches, and architectural-template-based design (which is the subject of this paper). The overwhelming majority of previous work in FPGA computing takes the first approach, focusing on manually-optimized ad hoc solutions [2]. Such approaches, although frequently achieving impressive performance, require significant hardware design effort and domain-specific knowledge, therefore

limiting their scalability, portability, and applicability.

The second approach, C-to-Gates, partly addresses these issues by providing application developers with a more familiar C-style language [3] in place of hardware description languages (HDLs) and relies on sophisticated CAD tools to efficiently map the application to a hardware platform. Recently, C-to-Gates research made significant progress and has resulted in numerous commercial solutions, notably AutoESL and Synfora. Unfortunately, almost all of these technologies, to our experience, require significant portions of existing applications to be rewritten, and usually with a different programming model in mind. Nevertheless, we believe that the C-to-gates approach holds great potential and in fact is quite complementary with our proposed MARC approach.

The MARC approach (the third route in Fig. 2) seeks to capture the strengths of the FPGA platform using a many-core architectural template without resorting to cumbersome HDL programming and time-consuming manual optimization. This methodology leverages computer architecture research, as well as modern compilers and C-to-Gates synthesis, to efficiently design structured high-performance many-core computer systems customized with application-specific knowledge. We hypothesize that this disciplined approach to FPGA computing with architectural constraints may win overwhelmingly in preserving hardware portability, reducing design effort, and possibly improving energy efficiency, despite its potential performance degradation relative to optimized FPGA solutions. In a larger perspective, we believe MARC to be a first step towards finding the right computational abstraction to characterize a wide range of reconfigurable devices, expose a uniform view to the programmer, and represent the computation in a manner that diverse hardware implementations can exploit efficiently.

B. Objectives

Our key objectives of this paper consist of (1) rethinking the design methodology of reconfigurable computing, (2) exploring ways to reduce hardware inefficiency between ASIC- and FPGA-based computing platforms, and (3) assessing the feasibility of creating an efficient, customized reconfigurable computing machine that incorporates ASIC-like efficiency but with much reduced design effort. To make our study concrete and evaluate our proposed MARC strategy quantitatively, we benchmark the performance of running a general-purpose Bayesian computing software on a MARC machine against that of a fully customized FPGA solution [1]. We choose general-purpose Bayesian computing because it captures many important algorithms in artificial intelligence and signal processing [4] and is thus widely applicable. Moreover, the algorithm contains a rich set of computational structures ranging from highly data parallel algorithms (individual node scoring) to control intensive ones (control graph propagation), making it a solid design driver for MARC.

II. Bayesian Computing Problem

A. Bayesian Graph Model

A Bayesian belief network is a probabilistic graphical model that represents a set of random variables and their conditional dependences via a directed acyclic graph (DAG). There exist efficient algorithms to perform inference and learning in Bayesian networks. Many important algorithms in artificial intelligence and signal processing can be expressed as operations over Bayesian probabilistic networks. The most prominent Bayesian learning algorithms include: the forward or backward algorithm, the Viterbi algorithm, the iterative "turbo" decoding algorithm, Pearl's belief propagation algorithm, the Kalman filter, and certain fast Fourier transform (FFT) algorithms.

Many important applications exist for the Bayesian graphical model ([4], Chapter 8) and consequently can be greatly accelerated by FPGA computing. Notable examples include early vision and DNA pyro-sequencing. Such problems can be optimally solved by by evaluating the marginal $p(x_{\mathrm{n}})$ on node n, defined by $p(x_{\mathrm{n}}) = \sum_{x_1} \cdots \sum_{x_{\mathrm{n}-1}} \sum_{x_{\mathrm{n}+1}} \cdots \sum_{x_{\mathrm{N}}} p(x_{1:\mathrm{N}})$ under the Bayesian framework. Unfortunately, a typical Bayesian network normally has an exponential number of such terms. As a result, even with speed-up techniques such as variable elimination, junction tree, and the sum-product algorithm, we still face prohibitively long evaluation time for any large-scale Bayesian network. Refer to [1] for more detailed discussions on various algorithms. This work focus on using FPGA-based computing to accelerate large-scale Bayesian networks evaluation for high throughput.

III. Many-Core Bayesian Computing

A. Many-Core Template

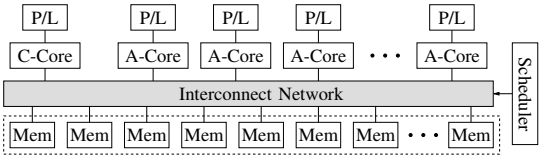


Fig. 3. Key components in a MARC machine. P/L—Private and local memory, C-Core—Control Processing Core, A-Core—Arithmetic Processing Core.

The overall architecture of a MARC system (Fig. 3) resembles a scalable many-core processor architecture consisting of a single C-Core (Control Processing Core) and multiple A-Cores (Arithmetic Processing Cores). Unlike embedded processor cores commonly found in modern FPGAs, although both C-Cores and A-Cores in MARC can be implemented as general-purpose RISC processor cores with a conventional five-stage pipeline [5], A-Cores can alternatively be synthesized as application-specific datapaths. Furthermore, the processing cores in MARC are completely parameterized with variable bit-width, reconfigurable multi-threading, and even aggregate/fused instructions. For example, in order to hide global memory access latency, improve processing node utilization, and increase the overall system throughput, a MARC system can employ fine-grained multithreading by injecting latency (via shift registers) and automatically retiming each core. Moreover, each processing core inside a MARC machine not only has a dedicated local memory accessible by its corresponding threads, but also can access a global, shared memory space implemented using block memories and off-chip DRAM (Fig. 3), accessible by all processing cores through the interconnect network.

B. Application-Specific Processing Core

The CAD flow of a MARC system consists of two separate tracks: 1) the compilation of application source code and 2) the high-level synthesis flow to generate customized datapaths or A-Cores specific to the target application. Both tracks are depicted in Fig. 4 and denoted by white and gray blocks respectively. Along the software track, the source

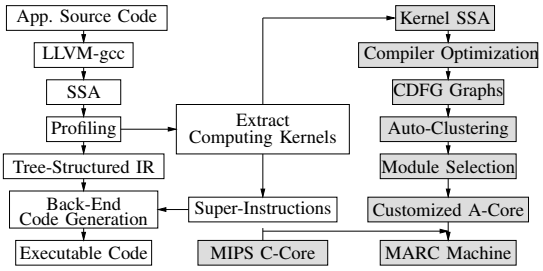


Fig. 4. CAD flow of executable code compilation and synthesizing application-specific processing cores.

code of target application in C/C++ is first compiled through `llvm-gcc` to generate ASTs (Abstract Syntax Trees)—a tree representation of the abstract syntactic structure of source code. Using a software interpreter, the `llvm` compiler calls a Call Graph Execution Profiler `gprof` [6] with `-gf` option to readily identify the most computationally intensive kernels by counting the number of pseudo-clock cycles in different code blocks. These code blocks are then promoted into super-instructions that encode all primitive operations contained in each code block. The rest of ASTs are then converted into a intermediate representation (IR)—a kind of abstract machine language that expresses the target-machine operations without committing to too much machine-specific detail. Unlike in the original `llvm` compiler framework [7] where single static assignment (SSA) is used as the intermediate representation (IR) before machine code generation, our modified MARC compiler uses the tree IR as suggested by [8], which proves to be more amenable to FPGA hardware mapping. Finally, both tree IRs and generated super-instructions are merged into a coherent binary file ready to be executed on a MARC machine.

Along the hardware track, SSA representations of kernels first go through generic compiler optimizations in `llvm`, such as dead code removal and constant propagation. The resulting SSA stream is then translated into a data structure called Control/Data Flow Graph (CDFG). Prior to hardware synthesis, the resulting CDFG needs to pass a two-step procedure: an auto clustering procedure that coalesces a group of neighboring nodes (typically in the form of subtrees) into a new node according to area and performance guidelines, and physically mapping the CDFG to the hardware components in a hardware library. Both steps proceed simultaneously and their underlying optimization is based on the well-known simulated annealing algorithm.

1) Control/Data Flow Graph (CDFG)

As in any synthesis system, the choice of intermediate data structure is essential. We chose mixed control/data flow graph (CDFG) [9] to represent the core algorithm as a flow graph with nodes connected with data or control edges. The nodes represent data operations, while the edges represent data precedences between those nodes. Unlike the task graph used in BCM [1], CDFG contains control edges to enforce extra precedence rules between nodes. Besides representing standard arithmetic operations as various operation nodes, the CDFG permits control-flow structures such as loops and if-then-else blocks often found in imperative languages. By introducing these control structures, CDFG can readily represent a hierarchical graph whose subgraphs represent the bodies of loops or conditionals. These subgraphs can subsequently contract into a single node at the next hierarchy level. Such hierarchical representation proves to be compact, descriptive, and efficient to store and manipulate the flow graphs. To illustrate the flexibility and effectiveness of the CDFG data structure, Fig. 5(a) depicts a small code snippet containing a `if` branch and a `while` loop. Fig. 5(b) and (c) show the CDFG before and after node combining respectively.

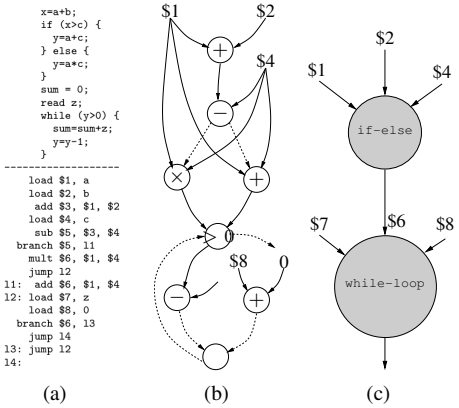


Fig. 5. (a) Source code snippet and resulting SSA. (b) Generated CDFG. (c) New CDFG with nodes after coalescing.

2) Auto-Clustering and Module Selection

Given a CDFG compiled from an target application, our next task is to choose hardware modules that minimize hardware cost, meet timing and throughput constraints, and accurately map the target application to the synthesized hardware. The difficulty of this selection is mainly due to the fact that when synthesizing reconfigurable hardware, there is a strong trade-off between hardware usage and computing performance. For example, in many repetitive operations such as `for` and `while` loops, the degree of hardware unrolling can significantly affect the overall performance. It is also well known that performing operations sequentially without intermediate registers can reduce the critical path delay. Therefore two operators connected through a register or memory should often be merged into a new operator. Moreover, pipelining functional units often incurs extra hardware overhead but can obtain noticeable performance benefits. Finally, in order to achieve high throughput, the pipelining has to be balanced with other components during module selection so that the available clock period can be used efficiently.

The synthesis procedure in our MARC framework is essentially a graph matching problem reminiscent of technology mapping with standard cells. It differs from a typical C-to-gates approach where compiled code IR (Intermediate Representation) is directly mapped to elementary gates. The key to the success of our arithmetic core synthesis is an extensive library of hardware components with diverse granularity ranging from primitive logic operators to algorithematic units such as floating point adders and multipliers. This work adopted two sets of library modules from the work of [10] and [11].

Given the number of parameters to consider, hardware module selection during synthesizing application specific A-Cores is clearly a NP-hard problem. It becomes even more complicated when considering timing constraints and functional units that use time multiplexing. To solve this multi-dimensional non-linear global optimization problem, we decided to use the well-known simulated annealing algorithm—a generic probabilistic meta-heuristic. The initial solution starts with all operations implemented on the cheapest available hardware and with full pipelining. Our synthesis engine then

clusters operations in a way that favors structures with high reusability, while simultaneously meeting correct timing constraints. During clustering, the synthesis engine may swap in more expensive, but faster, hardware for operations on the critical path. More implementation details of this technique can be found in [12]. Finally, the customized cores have the original function arguments converted into inputs. In addition, a simple set of control signals is created for cores to be started and to signal the completion of the execution.

C. Scheduling Super-Instructions on A-Cores

Scheduling super-instructions for the A-Cores on a MARC machine introduces new challenges absent from scheduling normal instructions in a microprocessor. Unlike a conventional microprocessor using registers to temporarily store intermediate computing results, a MARC machine, as discussed in Section III-D, possess no register at all. Instead, all private and local memory for both A-Cores and C-Core are implemented with Block RAM inside an FPGA device. As a result, each time two A-Cores need to communicate with data, they have to go through global memory. Moreover, intermediate data in a circuit can be transmitted by wire between modules, but only if the signal will not last multiple cycles or be transferred between multiple super instructions. In a MARC machine, different iterations of a loop behave like separate super instructions. This becomes an issue especially when implementing modulo scheduling. Finally, in a MARC machine, the cost metric of scheduling includes the number of A-Cores, local and private memory capacity, and the extra cost due to accessing global memory or exchanging data between two processing cores through the interconnect network. To our knowledge, there are no existing scheduling algorithms that can consider all these three cost components simultaneously. As a result, optimally scheduling super-instructions becomes almost infeasible.

Given limited hardware resources in a MARC machine, to maximize the overall throughput, the scheduler in MARC must determine which operations should be time-multiplexed, i.e., assigned to a single A-Core and executed sequentially, and which operations should be space-multiplexed, i.e., assigned to different A-Cores and executed in parallel. Obviously, if two super-instructions have data- or control-dependency, they can not start in the same cycle. It is desirable to assign two super-instructions with data dependency to the same A-Core, so that the intermediate results can be temporarily stored in local or private memory. If this is not possible, the start of these super-instructions must be separated by multiple cycles. Such delay may have an undesirable ripple effect and cause further delay for other subsequent super-instructions to start. Moreover, multiple accesses to the same block RAM must be spread out as constrained by data dependencies and memory communication bandwidth.

Given the above challenges in scheduling super-instructions, our modified `llvm` compiler uses two heuristic schemes for scheduling operations. Non-loop super-instructions are scheduled by a list-scheduling algorithm based on Graham's list-scheduling rule (LSA) [13], which guarantees a 4-approximation ratio to the optimal solution. For looping super-instructions and calculating the Initiation Interval (II) for pipelining the loop, we adopt the iterative modulo scheduling algorithm [14].

D. Memory Organization

Threads executing a kernel in a MARC machine can access three distinct memory regions: private memory, local memory, and global memory. Global memory permits read and write access to all threads within executing kernels on all processing cores. Local memory is a section of the address space shared by the threads within a computing core. This memory region can be used to allocate variables that are shared by all threads spawned from the same computing kernel. Finally, *private memory* is a memory region that is dedicated to a thread. Variables defined in one thread's private memory are not visible to another thread, even when they belong to the same executing kernel.

Physically, the private memory regions in a MARC system are implemented with LUT RAMs in an FPGA device, while local memory and part of global memory regions reside in the BRAMs. To allow a larger memory space, we also allow external memory to be used as part of the global memory region. To increase the number of global memory ports, we use both ports of each BRAM blocks separately, exposing each BRAM as two smaller single-port memories. Obviously, the achievable aggregate memory access bandwidth inside an FPGA is often far below its peak value, and the available amount of on-chip memory is small, even in comparison with a modern GPU. Nevertheless, the memory access mechanism in an FPGA is completely controlled by the user. This enables the MARC machine to consider application-specific access patterns in order to achieve high memory bandwidth.

IV. Hardware Prototyping

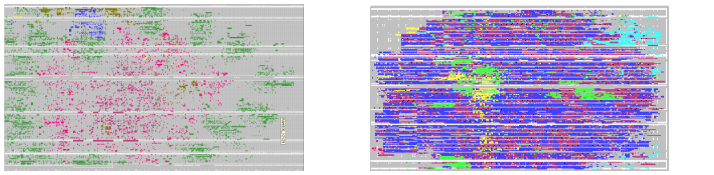
We implemented two prototypes of the MARC machine with a Virtex-5 FPGA (XCV5LX155T-2). MARC-I implemented both C-Core and 32 A-Cores as MIPS cores that support a subset of the MIPS ISA based on the Plasma CPU [15]. For hardware efficiency, A-Cores only support single-precision floating-point arithmetic operations, while the C-Core supports additional branching instructions. When implementing the MARC-II, we kept the design of C-Core intact, but implemented its A-Cores as automatically synthesized data-paths. The procedure by which the cores were generated is described in Section III-B. Each A-Core or C-core in the MARC-I and MARC-II machines are four-way multi-threaded to saturate the long cycles in the dataflow graph resulting from the target application and maintain a high throughput.

As in any computing system, memory access pattern significantly impact the overall performance of a MARC system. In the current MARC implementation, private or local memory accesses take exactly one cycle, while global memory accesses typically involve network dependent latency. The discrepancy between local and global memory accesses, we believe, provides ample opportunities for memory optimization and performance improvements such as caching, especially considering the hardware flexibility of MARC system manifested by application-specific processing core and customized interconnect network, which becomes pronounced when local memory accesses constitute the majority of all memory accesses as in our Bayesian computing problems.

Gigabit Ethernet is used to implement the communication link between the host and the MARC device; the available bandwidth may not be sufficient for larger applications, especially when using multiple MARC devices, where a

higher-bandwidth interface would be appropriate. To improve hardware utilization, we take advantage of the light-weight Ethernet interface design from the Plasma CPU [15]. Between processing cores and memory blocks, we leverage the GateLib project [16] from Berkeley to implement the crossbar switch and its associated arbiter. Because our design of the host-MARC interface is built using a latency-insensitive handshake, the physical transport can be readily replaced with other faster links such as PCI-E.

The hand optimized BCM implementation [1] is shown in Fig. 6(a), while the placed and routed MARC-II prototype in Fig. 6(b). Various main components in both platforms are color-coded. As shown in Fig. 6(a), constrained by long CAD tool run-time (about 20 hours), the hardware usage of our MARC implementation is approximately 90%, while the hand optimization BCM system in Fig. 6(b) utilizes about 68% of total chip resources.



16 Nodes (a) 1 C-Core + 32 A-Cores (b)

Slices: 17650, DSP Blk: 42, B. RAM: 4646 Slices: 20560, DSP Blk: 76, B. RAM: 5928

Fig. 6. Layout of (a) 16-node BCM and (b) 32-node MARC machine.

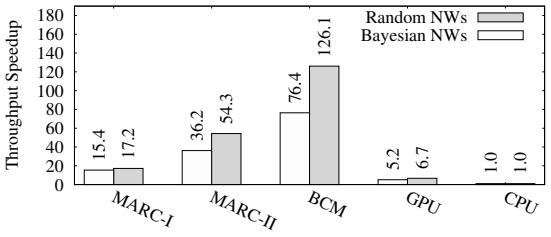
V. Performance Comparison

The number of floating point operations per second (GFLOPS) is used as the comparison metric for performance. We chose the sum-product (or marginalize a product of functions (MPF)) solver, often used for inference in very large Bayesian networks, as our benchmark application [17]. For MARC-I, MARC-II, BCM, GPU-based, and CPU-based, we run the identical MPF kernel source code from [17], while the GPU version of the MPF kernel runs on a NVIDIA GeForce 9400M graphics card, with 16 CUDA Parallel Processor Cores and 54 GFLOPS peak computing throughput. The CPU version is invoked on a single core of an Intel Core 2 Duo 2.4 GHz CPU with 32KB L1 and 3 MB L2 cache. Both the CPU and GPU versions are optimized for caching. When comparing run time, we report only the pure execution time, excluding the time for data transfers between the CPU and the GPU, for task graph processing, and for pipeline scheduling on CPU. To evaluate the performance on real MPF instances, we used Bayesian network instances generated from the real-life genetic data by SuperLink [18] [1] (Bayesian NWs). In addition, we evaluate some randomly generated Bayesian networks (Random NWs) for reference. In order to measure the size, or the input complexity, of each benchmark instance, we only consider the multiplications and summations required by the algorithm. In order to accurately measure the run time, we invoke the kernel on the same input until the accumulated running time exceeds five seconds as in [17], and then derive the time for a single invocation. Kernel invocation overhead ($\sim$10 μs) is ignored. The size of our benchmark designs vary from 0.001 to 1000 MFLOP; each case is run 10 times.

Table 7(a) presents the performance of all five platforms in GFLOPS for both randomly generated and experimental networks. As in [1], the BCM demonstrates an average 80x and 15x speedup over CPU and GPU solutions, respectively. The peak throughput of our BCM prototype is about 20.4 GFLOPS. Relative to GPU, MARC-I and MARC-II achieve on average 15x and 36x speedups respectively. More interestingly, we compare the performance of two MARC machines, without and with application-specific processing cores (MARC-I and MARC-II) respectively, against the manually optimized BCM FPGA solution [1]. We found that with synthesized application-specific A-Cores, MARC-II can achieve about 43% of BCM's throughput, whileas MARC-I can only achieve only about 15%. This discrepancy in throughput clearly shows the importance of application-specific core customization.

Platforms	Benchmarks					
	Random NWs			Bayesian NWs		
	Min.	Max.	Avg.	Min.	Max.	Avg.
MARC-I	1.12	5.55	2.56	0.69	4.28	3.28
MARC-II	**2.92**	**16.66**	**7.83**	**1.61**	**12.4**	**8.13**
BCM	11.25	29.23	18.23	5.78	20.4	17.31
GPU	1.09	0.23	0.96	4.94	0.18	1.15
CPU	0.50	0.12	0.14	0.34	0.11	0.22

(a)



(b)

Fig. 7. (a) Throughput achieved by all five platforms. (b) Speed-ups over CPU-based solutions achieved by MARC-I, MARC-II, BCM, and GPU-based for input Bayesian graph with variable sizes.

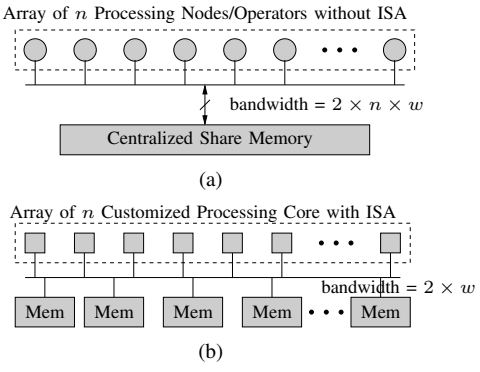
VI. Results Analysis and Modeling

This section presents an analytical performance model to understand BCM and MARC's performance discrepancy. More importantly, we hope such analysis can provide some insights about how to further improve the performance of MARC computing machine. We define f as the running clock frequency, n as the number of functional units, k as the number of elementary operators within each processing core, and μ as the IPC (Instruction per Clock Cycle) as defined in [5], respectively. Note that μ is often much less than 1 due to memory access stall, and can be computed as $\mu = \frac{1}{1+m}$, where m is the average memory latency amortized over each instruction. Finally, the throughput of a computing machine can be calculated as $T = f \times n \times k \times \mu = f \times n \times k \times \frac{1}{1+m}$. To differentiate BCM and MARC in their notations, we use subscripts B and M to denote them respectively.

The BCM [1] (Figure 8(a)) can avoid memory stalls and unnecessary pipeline bubbling by prior task graph processing, pipeline scheduling, and hazard-free memory allocation, which makes a BCM functionally equivalent to a multiprocessor with a high-bandwidth ($= 2 \times n \times w$) centralized shared memory. This memory bandwidth is infeasible for the current IC device technology. Furthermore, assuming all pipeline stages are busy and without any memory access hazards, the throughput of a BCM can be readily computed as $T_B = f_B \times n_B \times k_B \times \mu_B$. μ_B equals 2 because each operation core in BCM contains one floating point adder and one floating point multiplier. This scenario is drastically different from the MARC machine.

[1] Downloadable from http://bioinfo.cs.technion.ac.il/superlink/

Although a BCM machine's throughput can still be computed as $T_M = f_M \times n_M \times k_M \times \mu_M$, the μ_M value is far less than 1 and is measured to be 0.29 in our experiments, meaning that on average, each instruct needs to wait for about 2.4 clock cycles before its execution. Finally, to calculate the relative performance between BCM and MARC-II, $\frac{T_M}{T_B} = \frac{f_M \times n_M \times k_M \times \mu_M}{f_B \times n_B \times k_B \times \mu_B} = \frac{107 \times 3 \times 32 \times 0.23}{150 \times 2 \times 16 \times 1} = 62\%$, a bit larger than our measured performance data 43%. Our results have also shown that both BCM and MARC machines far out-perform the GPU solutions despite its formidable peak floating-point performance. We believe this is because the effective throughput for particular applications on a GPU is often far below its potential peak value, and is strongly influenced by branch prediction, cache management policy, and specific data access pattern, etc..



	f (MHz)	n	k	μ	m
BCM	150	16	2	1	1
MARC-I	74	12	$2+1$	0.36	1.8
MARC-II	107	32	$2+1$	0.29	2.4

(c)

Fig. 8. Abstract computing machine models for (a) BCM and (b) MARC. (c) Parameter values of BCM and MARC for throughput modeling.

Using the same strategy, we now explain why MARC-I can only achieve about 1/3 of MARC-II's performance. Again, we estimate the throughput of MARC-I. $\frac{T_{M-I}}{T_{M-II}} = \frac{f_{M-I} \times n_{M-I} \times k_{M-I} \times \mu_{M-I}}{f_{M-II} \times n_{M-II} \times k_{M-II} \times \mu_{M-II}} = \frac{74 \times 3 \times 12 \times 0.36}{107 \times 3 \times 32 \times 0.29} = 32\%$. From this analysis, it is clear that two factors contribute to the performance gain of MARC-II relative to MARC-I. First, because the A-Cores in MARC-II are much more hardware efficient than that of MARC-I, MARC-II has a much larger number of processing cores n. Secondly, because A-Cores in MARC-I are full-blown MIPS cores, there frequency f is about 30% lower than the frequency of A-Cores in MARC-II. As for lower μ value in MARC-I than that in MARC-II, we suspect it is because 1) smaller number of processing cores make instruction scheduling more optimal, 2) the construction of super-instructions inside MARC-II removes some temporary storage needed for intermediate results, therefore reduces the memory access bandwidth.

VII. Conclusion

Between optimized application-specific FPGA solutions and processor-based portable implementations, there exists a sizable and largely unexplored design space that holds the key to mitigating laborious HDL programming and relieving challenging hand-optimization. MARC offers a promising methodology that helps explore this design space. By combining a many-core architectural template, sophisticated logic synthesizing techniques, high-level imperative programming model, and state-of-art compiler technology [7], the MARC system can enable a user to harness the benefits of an FPGA platform without much of the required hardware design expertise. While we acknowledge fully that architectural constraints and high-level software abstractions can potentially erode the achievable performance of a reconfigurable computing system, our results show that such tradeoff is quite effective in the Bayesian computing domain: a MARC system achieves much of the performance exhibited by a manually optimized ad hoc implementation [1], but with a dramatic reduction in development effort and a significantly improved portability. Although the applicability of MARC to other problems, especially applications featuring little explicit parallelism and complex memory access patterns, remains to be investigated. we believe that MARC, the many-core approach to reconfigurable computing, introduces a new frontier to strive for energy-efficient high-performance reconfigurable computing.

References

[1] M. Lin, I. Lebedev, and J. Wawrzynek, "High-throughput bayesian computing machine with reconfigurable hardware," in *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 73–82, ACM, 2010.

[2] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, November 2007.

[3] Wikipedia, "C-to-HDL." `http://en.wikipedia.org/wiki/C_to_HDL`, Nov. 2009.

[4] C. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, August 2006.

[5] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2 sub ed., August 1997.

[6] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982.

[7] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[8] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Cambridge University Press, December 1997.

[9] S. Amellal and B. Kaminska, "Scheduling of a control data flow graph," pp. 1666 –1669 vol.3, may. 1993.

[10] G. Lienhart, A. Kugel, and R. Manner, "Rapid development of high performance floating-point pipelines for scientific simulation," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 182, 2006.

[11] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, and M. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," *International Conference on Field Programmable Logic and Applications*, vol. 0, pp. 317–322, 2005.

[12] S. Note, F. Catthoor, G. Goossens, and H. De Man, "Combined hardware selection and pipelining in high performance data-path design," pp. 328 –331, sep. 1990.

[13] A. Munier, M. Queyranne, and A. Schulz, "Approximation bounds for a general class of precedence constrained parallel machine scheduling problems," in *Integer Programming and Combinatorial Optimization, volume 1412 of Lecture Notes in Computer Science*, pp. 367–382, Springer, 1998.

[14] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 63–74, 1994.

[15] S. Rhoads, "The Plasma CPU," Sept. 2010.

[16] G. e. a. Gibeling, "Gatelib: A library for hardware and software research," tech. rep., 2010.

[17] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient computation of sum-products on gpus through software-managed cache," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, (New York, NY, USA), pp. 309–318, ACM, 2008.

[18] M. Fishelson and D. Geiger, "Exact genetic linkage computations for general pedigrees," *Bioinformatics*, no. 18, pp. 189–198, 2002.