Move Logic Not Data : A Conceptual Presentation

Ahmed Hemani, Muhammad Ali Shami
School of ICT, Royal Institute of Technology, KTH
Stockholm, Sweden
hemani/shami@kth.se

***Abstract*—Memory and global interconnect dominate the cost, power and performance of Embedded System on Chip (SOC) architectures. We contend that most of the architectural innovations being pursued today do not directly address these challenges. Move Logic Not Data is a scalable architectural concept where the movement of data is minimal. The logic that transforms/creates data is instead brought to the data. This is implemented with the help of Networks on Chip (NOC) which allows us to create seamless portioning of memory and logic resources. Two additional innovations, further improve the efficacy of this fundamental innovation: Reduce the code size. Movement of code also costs in terms of energy and latency. We propose using ultra complex algorithmic size instructions in the form of reconfigurable logic. Conceptual arguments and architecture that would implement these innovations are presented backed by theoretical analysis of their impact. Research challenges are identified that would need to be overcome to implement the proposed architecture.**

I. INTRODUCTION

Nomadic products host a suite of applications, dominated by high performance wireless communication and multi-media algorithms. The performance requirement of each of these applications can be extreme - the PHY layer of the now mundane 802.11a/g standard is 5 GIPs [1] and encoding decoding a h.264 stream in the meager CIF standard would need the ARM's flagship ARM11 processor to tick at 1.6 GHz with no cache misses, which is quite theoretical as it is not possible to clock ARM11 processors at this speed even in the latest 65 nm technology node. Meeting such extreme performance demands, for not one but a suite of applications and to power them on battery, produce them for mass market and keep the engineering cost manageable is an extreme SOC engineering challenge.

In SOC designs embedded memory is increasing primarily because of rapid increase in the amount of data handled by communication and multimedia algorithms to achieve higher bandwidth or resolution/quality [2].

The potentially arbitrary communication among applications forces system architects to often adopt a shared memory model of communication. To satisfy the large storage need, the cost and process factors, these products almost always having a single large external SDRAM memory. Concurrent applications that need high bandwidth memory access to the external SDRAM creates a bottleneck and results in usage of expensive L1 and L2 caches to hide latency and this explains the secondary need for memory. The architecture efficacy gap between the energy and performance needs of applications - communication, multi-media and security - and what is afforded by technology scaling is increasing [3]. Not only do the SOC architects have to contend with vastly large amount of data, they also have to tackle moving this data among applications at high-speed, a particularly difficult task in view of the well known fact that while transistors become fast with technology scaling, the global interconnect is not scaling [4] [5] as shown in Table I.

TABLE I. ALU, MEMORY AND INTERCONNECT DELAYS [4]

Operation	Delay in 130 nm	Delay in 50 nm
32b ALU Operation	650 ps	250 ps
32b Register Read	325 ps	125 ps
Read 32b across chip RAM	780 ps	300 ps
Transfer 32b acros chip (10mm)	1400 ps	2300 ps
Transfer 32b acros chip (20mm)	2800 ps	4600 ps

II. RELATED WORK

To close this gap between performance and memory a range of techniques have been deployed from the simple measures like increasing the clock frequency, increasing the depth of pipelining to more sophisticated measures like Instruction level parallelism (ILP), thread level parallelism (TLP) have been tried and the returns are diminishing [6]. ILP is primarily an architectural technique directed at improving the computational efficiency, and imposes more stringent demands on memory efficiency.

A. MPSOC

The latest architectural trend is the move to multi-processing. Advanced architectures are exploring the possibility of using NOC together with Multi-Processors to alleviate the bus bottleneck. Like ILP, we contend that the move to MPSOC is well justified but the goal is to overcome computational bottleneck, it does not effectively deal with the memory and the global interconnect challenge, which we argue is the central challenge.

To drive home the point that MPSOC does not alleviate the memory and interconnect challenge and are in fact plagued by it, consider the example shown in Figure 1, a slightly modified schematic of a wireless multi-media platform test chip from NXP [7]. This platform is a state of the art MPSOC in 65nm and a flagship product of NXP. To meet the large combined storage need of these applications and to enable arbitrary data communication among them, a large external Low Power DDR (LPDDR) memory is instantiated to implement a shared memory model of communication. As illustrated in Figure 1 with black lines, all the application processors need to access the external LPDDR creating a huge bottleneck. To hide the

latency, the processors have a sizeable L1 caches for instruction and data and potentially a system L2 cache. Other sub-systems have local buffers. Even a 1 GB/sec memory bus is barely able to sustain the worst case bandwidth requirement. Even with large, fast caches the processors - ARM1176 and Trimedia - typically operate effectively at one-third of their clocked frequency. In other words, if the ARM1176 is clocked at 400 MHz, the computational throughput is as if it was operating at 133 MHz with no cache misses. The inability to effectively handle large amount of data and large movement of data by this typical state-of-the-art architecture is the root cause of huge latencies, wastage of energy and silicon.

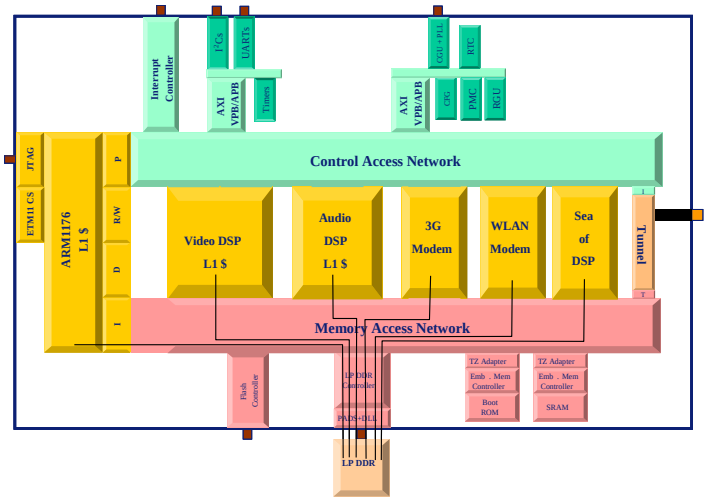

Figure 1. Memory Challenge In State of Art SOC

B. Processor In Memory

Memory is the central challenge has been recognized by a stalwart like Prof. David Patterson at Univ. of California Berkeley where a major project called Intelligent RAM (IRAM) has been launched. A vector media processor [8] for embedded systems is the first concrete outcome of this project. Solving the bottleneck to the external DRAM has motivated this work to incorporate on chip DRAM to get large bandwidth. While on chip DRAM in itself is not an innovation as it has been used by graphics chip designers in the past, the key contribution is to couple vector lanes to banks of DRAM via a fully connected inter-routing network in an architecture called VRAM. Recognizing that the fully connected network is overkill, a more optimized but less general version called CODE has also been developed. The focus of this project is to exploit the high on-chip memory bandwidth to fuel vector processor creating a complex on-chip communication network.

Flex RAM [9] is the name of an effort at Univ. of Illinois at Urbana Champaign that advocates Processor in Memory (PIM) approach to address the memory challenge. In this approach memory chips are replaced by 1 MB DRAM banks that includes a light processor. An on-chip interconnect connects 64 such memory bank processor together. Whereas IRAM targets embedded applications FlexRAM principally targets server applications. Flex RAM has not been commercialized mostly because the synchronization problem was not solved.

Imagine Stream Processor is another effort by Professor William Dally in university of Stanford to increase on-chip communication bandwidth in order to fuel many ALUs arranged in SIMD fashion [10]. In imagine processor, on chip memory called streaming memory and streaming register file is used to increase the communication bandwidth to 32Gbytes/s. Imagine processor utilizes the reference of locality principle and reduces the global bandwidth by having local buffers and stream register file. The intermediate results are stored in these local memories. Data movement between different kernels takes place through them as well instead of main memory, thus reducing global communication bandwidth requirement [11]. Once we have presented our architecture in detail in sections III and IV we will show the essential differences between imagine and MLND and show how this difference in moving logic vs moving data benefits both energy and performance.

III. MOVE LOGIC NOT DATA

The objective of this principle is to develop an architecture where the movement of data is minimal. The code/logic that transforms/creates data is instead brought to the data to transform it and/or create new data. Movement of code also costs in terms of energy and latency. We propose using ultra complex algorithmic size instructions in the form of reconfigurable logic. Traditionally, communication among applications/tasks has been achieved by either message passing or shared memory models. We propose a third alternative, a shared logic model.

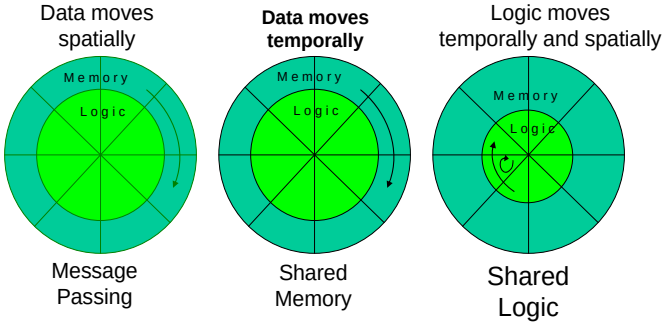

Figure 2. Visualizing the three models of communication.

Figure 2 provides an intuitive explanation of the differences between the three models. The inner circle represents logic and the outer circle represents the memory. Conceptually, the segments show the multi-processors hosting multiple applications and the alignment of segment shows a processor's association with a memory partition. In the message model, the inner logic circle does not rotate, the outer memory circle does, implying that logic segment once it has transformed data, the data moves and gets associated with another logic segment. In the shared memory model, the segments in outer circle represents temporal windows into a shared memory, at any particular time only one temporal window opens and the data is fetched and stored in local buffers of the logic segments. Generally, more than one temporal window could be open representing more than one shared memory. In the model we propose, it is the logic circle that rotates, while the memory circle stays stationary. The logic circle is shown in relatively

reduced size to underscore the fact that the code size is reduced. The reduction is both spatial - the code should take less space compared to the equivalent code in terms of assembly instructions and temporally - the code is changed less frequently as it represents ultra-large algorithmic size instruction. The logic also rotates in spatial and temporal sense. Spatial rotation implies that the same reconfigurable function can potentially move (its code) from one logic segment to another. Temporal rotation implies that different reconfigurable functions are sequentially loaded into the same logic segment.

IV. Quantitative Analysis

Consider an abstract chip shown in Figure 3(a), square in shape of side *a* units, dominated by memory with a small negligible area occupied by processor in the centre. Also assume that the memory is organized in *N* words. On an average these *N* words would travel approximately a*/2* distance. Now if we divide this chip into 4 equal parts and each part has its own processor in the middle. Assuming that the data in each partition stays within the partition and is only operated by the processor/logic in that partition, and then the average distance that these *N* words would need travel would be a/4. Generalizing, by dividing the memory into 'n' partitions, we reduce the average distance travelled by each of the *N* words by factor $n^{1/2}$ compared to the original un-partitioned case. Latency is dominated by interconnect and memory access. The delay in interconnect is directly scaled down by $n^{1/2}$ and also its switching capacitance. Since the delay in memory is related to its size by $(SIZE)^{1/4}$ [12], the memory delay scales down by $n^{1/4}$. Let $L_{Total} = L_c + L_i + L_{m,}$ where L_{Total} is the total latency, L_c is the compute latency, L_i is the interconnect latency and L_m is the memory latency. Further assume that $L_i = 10L_c$ and $L_m = L_c$. This is partly based on data shown in Table I. Then $L_{Total} = 12\ L_c$. Now if $L_{Total(n)}$ represents the Total Latency for the n-partitioned case

$$L_{Total(n)} = L_C + \frac{10L_C}{n} + \frac{L_C}{\sqrt[4]{n}} = L_C \frac{n + 10 + n^{3/4}}{n} \quad (1)$$

$$\frac{L_{Total}}{L_{Total(n)}} = \frac{12n}{n + 10 + n^{3/4}} = S \quad (2)$$

Where S is the scaling factor, which is the ratio of total latencies for the un-partitioned and partitioned case. Now we use the above arguments to see its impact on dynamic power consumption; the partitioning does not have any impact on static power consumption under the assumption that there is no change in the total area of the chip. The dynamic power consumption for the unpartitioned case is $P = CV^2f$. Now if we consider dynamic power for a single partition $P(1)$ in the n-partitioned case, the switching capacitance scales by *n* and since the Total Latency $L_{Total(n)}$ has gone down by a factor S, we can scale down the operational frequency by S to maintain the same throughput.

$$P(1) = \frac{C}{n} V^2 \frac{f}{S} \Rightarrow n.P(1) = CV^2 \frac{f}{S} \quad (3)$$

$$\frac{n.P(1)}{P} = \frac{n + 10 + n^{3/4}}{12n} \quad (4)$$

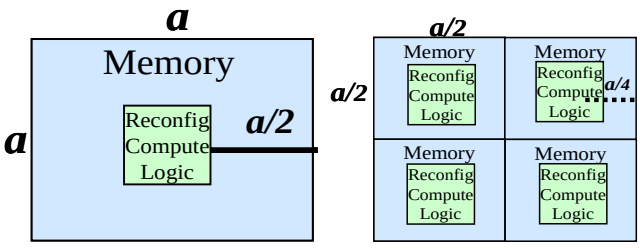

Figure 3. (a)An embedded soc (b)Embedded soc partitioned into 4 parts.

Equation 4 is the factor by which the dynamic power consumption goes down for the n-partitioned case and is shown in Figure 4. The above analysis though broadly accurate does not factor in the fact that with the scaling frequency, the V_{dd} would also scale down. Besides partitioning is a key complication. Getting a clean partitioning is a non-trivial problem and beyond a certain partitioning, the inter-partition communication will start to eat into the benefits of partitioning.

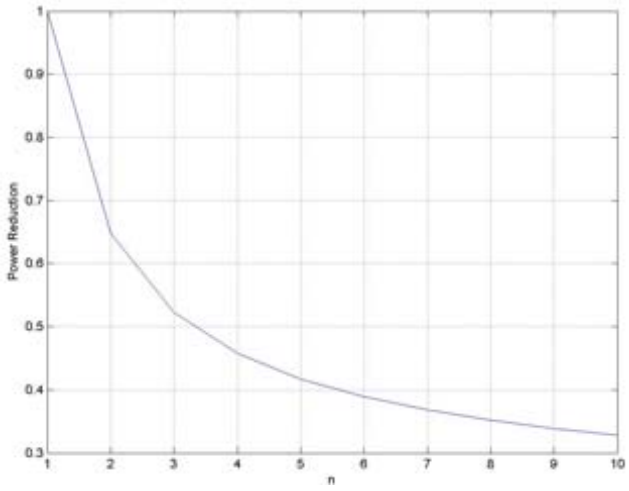

Figure 4. Power scaling as function of n

V. MLND Architecture

A conceptual diagram of the MLND architecture is shown in Figure 5 which is composed of the following components:

A. *Memory Pool*

Memory Pool is a pool of runtime partition-able memory. Each partition has the capacity to hold the dataset required for an application. While the MLND architecture provides hooks for implementing the partition, it is the software that characterizes an application for its memory needs based on bounded use case statements and manages the partitioning at runtime. Two kinds of memories are used. One that has higher bandwidth and relatively low capacity is meant for use by the physical (PHY) layer of the seven layer OSI model and the

other that has relatively lower bandwidth but higher capacity is meant for the upper six layers of the OSI model.

B. Arithmetic Logic Pool

Arithmetic logic Pool is a pool of runtime partition-able reconfigurable arithmetic logic. This logic will be glued together to implement complex integer units to implement MACs, Butterflys etc. Like the memory partition, the arithmetic logic partition is dimensioned to fulfill the needs of an application. Depending on the performance constraint, it is the MLND compiler's task to determine the degree of parallelism, algorithmic level pipelining etc.

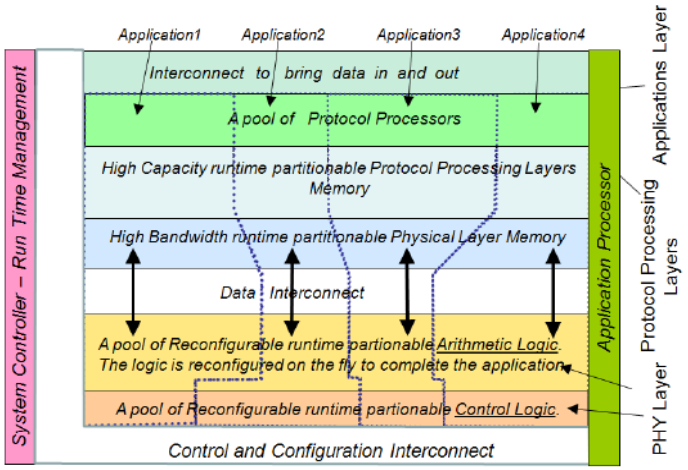

Figure 5. Conceptual view of Move Logic Not Data Architecture

C. Control logic

Like arithmetic logic, the pool of control logic provides the possibility of creating FSMs on the fly to control the arithmetic data path and memory operations. Essentially, the created FSM works like the hardware FSM and together with the arithmetic logic comes very close to the hardwired ASIC model of implementation, albeit with an overhead for some generality and the ability to partition. The separation of arithmetic and control logic is a key innovation to gain generality while maintaining efficiency and performance. This will be achieved by composing concurrent FSMs from a library of templates corresponding to various computational behaviors. The Arithmetic and Control Logic Pools together are dedicated to implementing the PHY layer of the OSI model.

D. Protocol Processor Pool

In the OSI model, the five immediate layers above the physical layer are characterized by control and memory intensive functionality and their memory access pattern is very irregular as compared to that of PHY layer so they are ideally served by a protocol processor. We intend to run the Application layer on a separate Application Processor. The protocol processors have access to high capacity memory and also control the transfer of data between the high-bandwidth PHY layer memory partitions and the Protocol processor memory partitions.

E. Interconnect

The NOC based interconnects implemented in MLND will give the flexibility to seamlessly partition the system. This kind of partitioning will make custom ASIC processors on the fly with its own memory unit, interconnect and data path The MLND architecture will have three kinds of NOC based interconnects as shown in Figure 6.

External data NOC brings external data into the chip and deposits it into the right memory partition and then once it has been processed takes it out to external memory. This kind of interconnect requires speed and flexibility and will be made up of high speed packet switched NOC.

Data NOC couples the memory partition to the arithmetic and control logic partition and it is this interconnect that MLND ensures is qualified as local interconnect. This interconnect will connect some memory to some ALU for a complete reconfigurable cycle and the interconnection will remain fixed for that reconfigurable cycle. This requires less flexibility, so we can implement this interconnect with high speed circuit switched NOC.

Control and Configuration NOC is used to control and configure the partitions and for operation and maintenance. This interconnect don't require much bandwidth but do require flexibility. So low speed packets switched NOC will be implemented for this kind of interconnect.

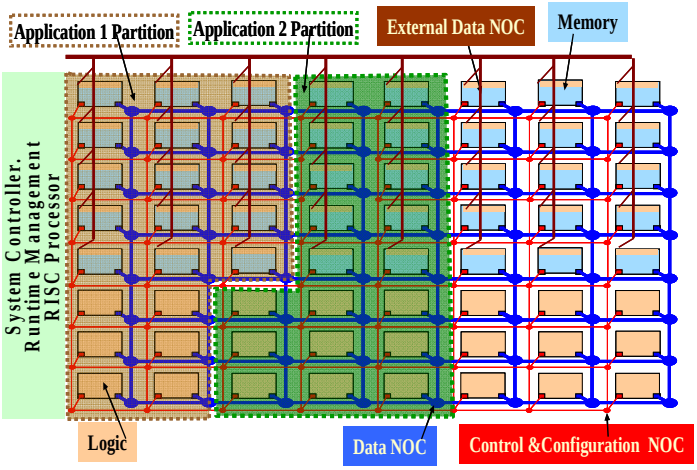

Figure 6. Conceptual view of Partitioning using NOC

F. System/Application Controller

The system controller provides the runtime management services of allocating memory and logic/arithmetic/protocol processor pools and partitioning. Figure 5 shows two RISC processors, System Controller and Application Processor, flanking the entire MLND structure. One of them is intended as a systems controller and the two would share the application layer functionality of the applications. The choice of two RISC processors is arbitrary at this stage; the actual number will be the outcome of the dimensioning of the MLND architecture by the design tool proposed as a research topic in this project.

G. Run Time Management

The MLND runtime support provides interface to the external world, manages resources in the MLND architecture and co-ordinates execution of the applications. The MLND Runtime System is conceptually made up of three interacting components. The External Interface manager interacts with the external world. This involves interrupts that trigger applications and peripherals responsible for exchange of data between external world and the MLND system. Resource manager, as the name suggests manages the resources like memory, arithmetic and control logic, bandwidth and energy/battery. When an external signal, a touch screen, a jog dial or a radio signal triggers an interrupt, the External Interface Manager passes on the request to the Resource Manager. The Resource Manager in turn analyzes the available resource, makes an allocation and passes on the constraints and requirement to the Application Manager. It is the Application Manager that instantiates the controllers: the application controller, the protocol processing code and the PHY layer controllers based on the constraints received from the Resource Manager. When an application is complete, it informs The Resource Manager, via the Application Manager resulting in an update of the available resources.

Besides the physical resources, the other key resource that the MLND Runtime Manager would have to handle is that of energy. The MLND architecture, from grounds up is built to implement the philosophy, if a resource is not being used, keep it shut. The other key energy management principle is to run the application at the optimal voltage frequency operating point, using the Dynamic Voltage Frequency Operating principle. The MLND architecture will introduce the novelty of having dynamic voltage islands and the RTM will play a critical role in its management

H. Methodology

MLND programming methodology would map a suit of applications, typified by modems and codecs, to the MLND architecture. While the details of the Design Environment (DE) are the objective of the proposed research, the conceptual steps and the components involved in the mapping process are shown in Figure 7.

1) Step 1. System Partitioning

In this step, the MLND DE partitions the application into three sets of functionality, each intended to run on a different kind of compute engine adapted to the nature of the functionality. The Application Layer on a RISC processor, the next five protocol processing layers on a customized protocol processor and the Physical layer on the reconfigurable arithmetic/control logic tiles. As a result of this step, we get the total application partitioned into the Application layer, the Protocol Processing Layers and the Physical Layer. These three partitions now communicate using the NOC based interconnect structures of the MLND architecture.

2) Step 2. System Dimensioning

This step identifies the overall storage need of the application, dimensions it and budgets the energy and performance constraints among the different partitions. We rely on the fact that the MLND concept primarily targets DSP oriented applications, where the nature of functionality, characterized by isochronous traffic, and the standards specification (from IEEE, ITU, MPEG etc) considerably helps gauge the storage needs. Further narrowing the search space are the architectural constraints that are imposed by the MLND architectural template and the memory technologies available. Lastly, the way application is modeled in terms of its data organization and access to it has a strong bearing on the storage dimensioning. The result of this step is the critical decision on what parts of dataset will be on chip and off-chip. Also decided in this step are the dimensions of the high bandwidth PHY layer memory and the high capacity Protocol layers memory. These decisions are also key inputs to the run time management system that needs to implement these partitions.

After the Steps 1 and 2, it should be possible to create a transaction level simulation model of the application, where the different partitions and storage interact using the MLND control, configuration and interconnect infrastructure. The next steps would refine the individual partitions.

3) Step 3. Protocol Layer Compilation

This step refines the protocol processing layers. The implementation style is software running on a RISC processor enhanced with custom instructions and internal memory structure to optimize the energy and performance of the protocol processing functionality. Creating such ASIPs (Application Specific Instruction Processors) is well researched with MESCAL [13] methodology as a prime example of this and will be the basis for this step. The logic for enhancing the RISC processor is built from reconfigurable tiles that can be configured to do various protocol processing functions. This makes the protocol processing pool homogenous and gives the runtime system freedom to place an application anywhere as required by the dynamic runtime situation.

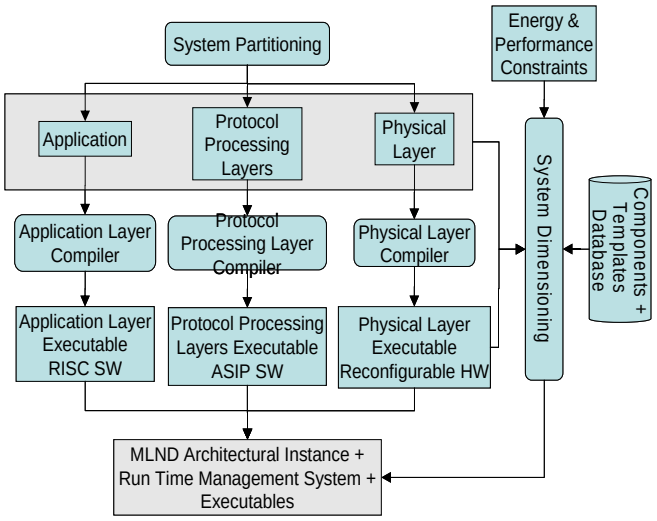

Figure 7. The MLND Design Flow

4) Step 4. Physical Layer Compilation

This step dimensions and instantiates the reconfigurable compute engine composed of arithmetic and control logic tiles. The key insight behind this step is that the physical layer functionality in most cases is composed of standard DSP

functions whose architectural implementation space is pretty well understood. Examples of such functions are FFT, Viterbi, FIR filters etc. This understanding of architectural space is captured as templates and used to narrow the design space that the synthesis tool would otherwise have to search. For the few functions that do not have templates, we intend to use existing High Level Synthesis tools to create an implementation.

The results of this layer are the configuration codes that implement the arithmetic and address generation parts for the different algorithms/functions and the configuration codes for the control logic to implement the Finite State Machine (FSM) that controls the arithmetic and address generation logic. This layer also synthesize the PHY layer control and memory logic that glues together individual algorithms/functions that make up the PHY layer. This controller is again implemented as an FSM, and controls the individual algorithmic level controller synthesized in steps 3 & 2. More importantly, this controller is responsible for controlling the pipeline decision.

An additional key aspect regarding evaluation and estimation in the Steps 3 and 4 is that the MLND is an architecture built using regular tiles, and these tiles are built using full-custom macro implementation styles, this leaves little room for uncertainty in wiring delay as is common using the logic synthesis/standard cell based methodology. This approach, we believe will be key to our ability to achieve not only the best energy / performance metrics, but also the regularity of layout makes it possible to predict the energy and performance.

5) Step 5. Application Layer Compilation

In this step, the application layer is compiled to the RISC based application processor. While compilation to the RISC processor is straight forward, the application layer is essentially a controller that interfaces to the Protocol Processing layers via the MLND architectural elements. The Runtime Management System interacts with the Application Controller as the main agent for activating an application and knowing when an application is complete.

VI. Comparison with Previous Work

MLND is a flexible, scalable general purpose architecture which is not only suitable for nomadic products but equally suitable for high performance computing systems like base station and super computing. The computational requirements for a nomadic product may change with time and depends on the usage. For instance a nomadic user at times may be using MP3 player and browsing, at some other time he may only be using it as a GSM/3G phone, doing a video call and at most of the time the phone is sitting idle. So a nomadic product like cell phone requires flexibility so that it could offer the required computational power according to user's need and switch off/on additional resources.

MLND is a natural candidate for such a requirement. This ability to create runtime partitions of memory, arithmetic and control logic to implement custom ASIC like macros are the key to implementing the MLND theme: once the data come into the memory partition, the reconfigurable logic (arithmetic and control and protocol processing) implements a succession of algorithms to transform the data. The architecture guarantees that the memory partition and the arithmetic and control logic partitions are geometrically close enough that they qualify as being connected via local interconnects that does scale with technology as opposed to the global interconnects that do not scale with technology[5].

In MLND the separate control logic controls a set of partitioned memory and logic blocks called cluster of memory/logic. The partitioning will be done by using NOC [15]. Kernels will be implemented on this cluster of memory/logic. This kind of partitioning will make custom ASIC processors on the fly with its own memory unit, interconnect and data path. The data path unit will be parallel or serial as required. These custom processors will act like multi core/multi processors, exploiting (Instruction Level Parallelism), or DLP (Data level parallelism), algorithm level pipelining where all algorithms can be executed concurrently, and working in a pipelined fashion. Every cluster has its own individual control which makes it possible to clock them at different clock frequency, hence implementing dynamic voltage frequency scaling techniques to reduce power or switch them on off by resource manager.

The basic theme of MLND is to keep the wire distance between memory and logic minimum (local wire) so that the power consumption on interconnect is very small. In traditional architectures, ALUs are fueled by feeding data from memories which are far from them; hence dissipating a lot of power in interconnects. Such architectures do not scale with technology. In MLND the logic close to data memory is re-programmed to perform operation on the data stored in that memory. Imagine processor [10] [11] is also designed keeping interconnect power consumption in mind and is closest to MLND theme. A comparison of data flow of OFDM in MLND with Imagine processor is shown in Figure 8.

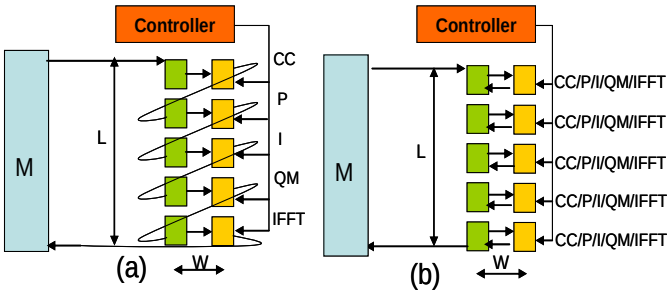

Figure 8. (a) Imagine Processor (b) MLND

In Imagine processor data enters into the first logic block i.e. Convolutional Coding (CC), processed and then saved into the memory as shown in Figure 8. From there it goes to Puncturing (P), then back to memory. It goes to Interleaving (I), QAM Mapping (QM) and IFFT in the same way before going to the main memory. The total logical distance travelled by the each data word, in case of Imagine processor, is L+10W. Assuming the arithmetic blocks are of same dimensions, in case of MLND the kernels are reconfigured instead of moving the data; reducing the total distance travelled by each data word to 10W. An OFDM symbol uses 64point FFT. A DVB

standard uses 2048 points FFT. A reconfigurable DPU designed for MLND take 3-bits to configure. A radix-4 FFT butterfly uses 14 such DPUs. Suppose Z is the energy consumed by single bit to travel L distance shown in Figure 8. The Imagine processor configures the data path once and keeps its state for the life time of the application. On the other hand, MLND reconfigures the data path after a certain reconfiguration time. The number of bits needed to reconfigure the data path, travel on average L/2 distance. Assuming the arithmetic block of same dimensions, the energy comparison between moving data and code is done in Table II; which shows that it takes more energy to move data then moving the code as code size is much smaller then data size. Table II shows results for just one sample of FFT. Of course the hardware will operate on many more samples before undergoing reconfiguration which further confirms that code movement is cheaper then data movement in terms of energy. One may argue that code movement will be global and data movement will be local. According to [5] in 65nm global wires are 10 times slower then local wires, but the data in the Table II shows that energy for movement of code, in case of 64 point FFT, is 100 times less then energy required for movement of data. The figures are even better for 2048 point FFT.

TABLE II. ENERGY PER BIT FOR DATA MOVEMENT IN IMAGINE PROCESSOR VS CODE MOVEMENT IN MLND

FFT	Energy per bit for Data Movement	Energy Per Bit for Code movement
OFDM 64 points	64x16xZ=1024Z	14x3x5xZ/2=105Z
DVB 2048 points	2048x16xZ=33554432Z	14x3x5xZ/2=105Z

Minimizing the movement of data at PHY layer is a good thing but not sufficient. Because huge movement of data also happens at MAC layer and if that is left un-addressed the solution as a whole will still suffer from performance, energy and cost in-efficiencies. That is where the protocol processing layer takes over takes to minimize the data movement.

VII. CONCLUSION

MLND is an energy aware, scalable architecture, which minimizes the data movement inside the chip hence reducing power consumption. It has a regular structure and can be implemented in full custom. Ability to know exact wire lengths because of full custom implementation, and energy aware mapping and runtime system, makes it significantly different and better from the competitors. Traditional architectures lack this ability. MLND is flexible enough to be used in nomadic products as well as high end computing systems and super computers.

REFERENCES

[1] End to End Reconfigurability White Paper. Hardware Technology Exploration: Impact of Technology Evolution on End to End Reconfigurability. http://e2r.motlabs.com/whitepapers/E2R_WhitePaper_HardwareTechExploration_December05.pdf

[2] Erik Jan Marinissen, Betty Prince, Doris Keitel-Schulz, Yervant Zorian. Challenges in Embedded Memory Design and Test. Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05).

[3] Panagiotis Tsarchopoulos. Objective ICT-2007.3.4 Computing Systems. FP7 Information Day. March 7, 2007. Brussels.

[4] Jan Rabaey. Silicon Architectures for Wireless Systems. Tutorial Part 1. Hotchips Conf. 2001.

[5] Bjerregaard T. and Mahadevan S., "A Survey of Research and Practice of NoC", ACM Inc. New York, USA, 2006.

[6] Michael J. Flynn, Patrick Hung, Kevin W. Rudd. Deep-submicron Microprocessor Design Issues. IEEE Micro. July-August 1999.

[7] Hemani, A, Klapproth, P. Trends in SOC Architectures. Chapter in the book "Radio Design in Nanonmeter Technologies" Editied by Professor Mohammed Ismail and Delia Gonzales. Springer Verlag 2006/2007

[8] Christoforos Koszyrakis. Scalable Vector Media-processors for Embedded Systems. PhD Thesis. Univ. of California Berkeley. May 2002. Report No. UCB/CSD-02-1183

[9] Kang Yi el.al. FlexRAM: Toward more Advanced Intelligent Memory System. Proceedings ICCD Oct. 1999.

[10] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, and Andrew Chang, "Imagine: Media Processor With Stream," IEEE Micro, March/April 2001, pp. 35-46.

[11] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter Mattson, and John D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," Proceedings of the 31st Annual International Symposium on Microarchitecture, Nov. 30 - Dec. 2, 1998, Dallas, Texas, pp. 3-13.

[12] Michael J Flynn, Patrick Hung. Microprocessor Design Issues: Thoughts on the Road Ahead. IEEE Micro. May-June 2005.

[13] J. Rabaey: Reconfigurable Computing: The Solution to Low Power Programmable DSP; Proc. ICASSP'97 Munich, Germany, April 1997.

[14] Marinissen Erik Jan, Prince Betty, Shultz D.K, Zorian, Yervant. Challenges in Embedded Memory Design and Test. Proceedings of DATE 2005.

[15] A Hemani, A Jantsch, S Kumar, A Postula, D Lindqvist, J Öberg, M Millberg. Networks on Chip: An architecture for the Billion Transistor Era. Proceedings of the IEEE Norchip Conference. October 2000.

[16] Reiner Hartenstein. Coarse Grain Reconfigurable Architectures. Proceedings of Asia South Pacific Design Automation Conference. 2001. Yokohama, Japan