# Does $Q = MC^2$?

## (On the relationship between *Q*uality in electronic design and the *M*odel of *C*olloidal *C*omputing)†

Radu Marculescu, Diana Marculescu

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

***Abstract:*** *This paper introduces colloidal computing as an alternative to the classical view on computing systems in terms of design feasibility, application adaptability and better energy-performance trade-offs. In colloidal computing, simple computational particles are dispersed into a communication medium which is inexpensive, (perhaps) unreliable, yet sufficiently fast. This type of clustering into computationally intensive kernels with loose inter-particle communication, but tight intra-particle communication is typical not only for the underlying hardware, but also for the actual application which runs on it. We believe that the colloidal model is appropriate to describe the next generation of embedded systems. For these systems, a significantly better design quality can be obtained via run-time trade-offs and application-driven adaptability, as opposed to classical systems where optimizations are sought in a rather statical manner.*

**Keywords:** Embedded systems, platform-based design, asynchronous design, power dissipation, colloidal computing.

## 1. Introduction

With increasing levels of integration, many functional blocks and modules can be put together on the same chip. However, with growing die sizes and shrinking cycle times, clock skew, as well as power consumption due to clock nets and buffers have become major problems. To compensate severe clock skew, one possible solution put forth is the Globally Asynchronous, Locally Synchronous (GALS) design paradigm [14]. This methodology relies on having synchronous blocks that communicate asynchronously one to another, thus reducing the need of a global clock and alleviating the clock skew and global clock power problems.

It is our belief that this new design paradigm is an intrinsic part of a more general model which supports *local computation* and *inexpensive communication* among computational elements. We call this the *model of colloidal computing* ($MC^{2a}$): simple computation particles are dispersed in a communication medium which is inexpensive, (perhaps) unreliable, yet sufficiently fast. This type of clustering into computationally intensive kernels with *loose inter-particle communication*, but *tight intra-particle communication* is typical not only for the underlying hardware, but also for the actual application that runs on it.

In this paper, we propose and discuss the applicability of the $MC^2$ - at both behavioral and microarchitectural levels - and provide a different perspective to the classical view on computing systems in terms of design feasibility, application adaptability and better energy-performance trade-offs. Specifically, by clustering the application into computation intensive kernels, the mapping on application-specific platforms can be done much more efficiently. The architecture can reflect the natural application clustering, and include additional constraints such as power consumption, clock skew or manufacturability-driven die size constraints which are directly related to the quality of the electronic design [2]. At an even finer grain, different components (such as programmable modules or core processors) can embed a colloidal model at the architectural/microarchitectural level. What we have seen over the years are examples of *static aggregation* of computational resources, be it at application, architecture or microarchitecture level. Static aggregation is thus the result of application or architecture partitioning for achieving minimal communication overhead and maintaining different quality metrics (such as performance, energy or fault-tolerance) within certain limits. However, it is our belief that, aggregation of computational elements can also be done *dynamically* (or adaptively) at *run time*. Specifically, for achieving high quality in modern electronic systems, we need to move the "aggregation" process at run time, by actively re-configuring the architecture and application for better adaptability to the operating environment.

From a very abstract point of view, the behavior of computing systems is essentially the result of interaction among three fundamental entities: application, architecture and communication (Fig.1). The interplay between hardware and software with respect to these three entities determines the overall operation of these systems under different design and/or run-time constraints.

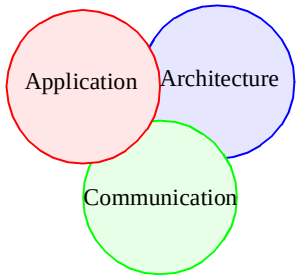



Fig.1 Fundamental entities in computing systems

In what follows, we intend to illustrate the fractal nature of colloidal computing and discuss its implications at both system and microarchitecture level. Our main goal is to point out some interesting directions for research rather than present detailed solutions to particular problems. To this end, we intend to introduce, in an informal manner, a new model which is able to describe the next generation of embedded systems. For such systems, a significantly better design quality is obtained via run-

a.*Colloid* [käl´oid] = a substance consisting of very tiny particles (1 nm and 1000 nm) suspended in a continuous medium, such as a liquid, a solid, or a gaseous substance [1].

† This research was supported in part by Semiconductor Research Corporation under grant 2001-HJ-898.

time trade-offs and application-driven adaptability as opposed to the case where optimizations are sought in a statical manner.

## 2. Colloidal computing

Our proposed colloidal computing model supports local computation and inexpensive communication among computational elements. We call such a model colloidal computing ($MC^2$) by making an analogy to surface phenomena in physical chemistry [1]: simple computational particles are dispersed in a communication medium which is inexpensive, (perhaps) unreliable, yet sufficiently fast (Fig.2). This type of clustering into computation intensive kernels with *loose inter-particle communication*, but *tight intra-particle communication* is typical not only for the underlying hardware, but also for the actual software application that runs on it.

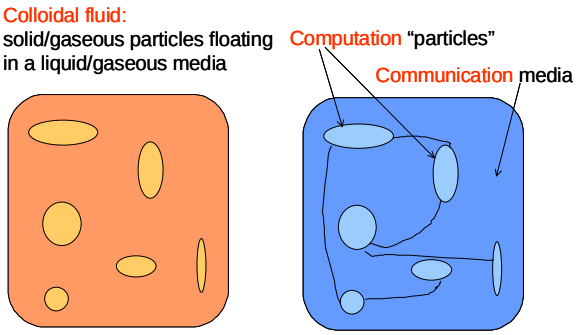



Fig.2 Colloidal computing

We will show in the sequel how this model applies to different hardware/software systems, exhibiting various trade-offs between flexibility, complexity and performance. This claim is supported by the following key observations:

- In the case of general purpose processor-based systems, the view of a globally clocked, monolithic architecture has started to change. As discussed in [3], one possible solution to coping with the clock skew and process variation issues is to switch from a single clock to multiple clock domains, communicating or synchronizing only when they need to exchange information. Even in the single-clocked high-end cores, with increasing die sizes, clock skew management becomes a challenging design problem. A possible solution to this is to move some of this effort from design-time to run-time, by allowing larger, but bounded clock skews, managed by handshaking mechanisms [4,5]. We will show that a typical pipelined core processor lends itself to a "tight local computation/loose global communication" design paradigm, in line with the proposed colloidal model.
- A large class of *software* applications is characterized by high spatial and temporal locality. In fact, the set of media processing applications are characterized by highly computation intensive kernels that loosely communicate one to another. Most applications exhibit a natural clustering into computational kernels that compute locally and communicate infrequently to exchange results. Such kernels (typically associated with media or signal processing applications) can thus be mapped on separate computational particles that communicate loosely.
- The underlying hardware platform can be as simple as a single-embedded processor-based system, or as complex as a multiple-core system on the same chip. To be able to achieve high integration and functionality on the same die, communication has to be achieved via *on-chip communication* networks instead of using a globally synchronous clock for the entire chip. Thus, computation has to be as *localized* as possible, with loose and inexpensive communication among different blocks or tiles on the same chip [6]. The architecture can reflect the natural application clustering, or include additional constraints such as manufacturability-driven die size constraints, clock skew or power consumption.

We propose to apply $MC^2$ to both application software *and* underlying architecture. As it will be seen in the sequel, $MC^2$ occurs in both hardware and software sides of the world, either in the form of GALS platforms or cores, or in the form of explicitly concurrent applications. Similar modeling at both architectural or application levels, as well as the microarchitectural level can be instrumental in the step of mapping an application on a specific platform during the design process.

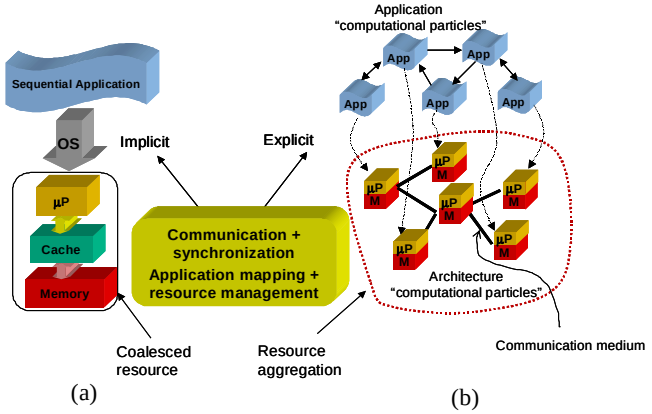



Fig.3 Coalesced vs. aggregated resources

To show the specifics of the colloidal computing model, some observations are in order. In case of unstable colloidal suspensions, colloidal particles tend to coalesce or aggregate together due to the Van der Waals and electrostatic forces among them. *Coalescing* reduces surface area, whereas *aggregation* keeps all particles together, without merging them. Similarly, in the classical computing system, there is a clear separation between computation and communication with the outside world. The resources of a classic system are *coalesced* together in a monolithic form (Fig.3a), as opposed to the case of $MC^2$ where useful work can be spread among many, small, (perhaps) unreliable computational elements that are dynamically *aggregated* depending on the needs (Fig.3b).

We point out that aggregation of computational particles can be done statically at *design time*, or dynamically (adaptively) at *run time*. The equivalent of electrostatic forces or Van der Waals forces [1] in the computing world is given by the naturally local computation inside computational kernels, as well as the *decoupling* between them, be it at application, architectural or microarchitectural levels. As an analogy, in case of unstable colloids, a minimal energy configuration is achieved via coalescing (e.g., oil in water) or aggregation (e.g., polymer colloids). In $MC^2$-based systems, a "stable" configuration is one that achieves the required functionality, within prescribed limits for performance, power and probability of failure. We propose to employ aggregation (Fig.3b) (or *dynamic connection*) of computational "particles" based on their state (i.e. functional or not; idle or active) and their probability of failure so as to

achieve a required quality of service (QoS) (characterized by performance, power and fault-tolerance). In addition, the mapping of computational kernels to the computational particles is also dynamically adaptable, depending on the needs. The re-organization and re-mapping can be achieved by *thin middleware clients*, sufficiently simple to achieve the required goals without prohibitive overhead. Static aggregation is thus the result of application or architecture partitioning for achieving minimal communication overhead and maintaining quality metrics (performance, energy or fault-tolerance) within certain limits. On the other hand, *dynamic* (or *adaptive*) *aggregation* is explicitly done on-the-fly whenever operating conditions change (e.g., failure rate is too high or battery level is too low), thus moving the application mapping and communication architecture configuration process at run-time.

In the following, we discuss how the colloidal computing is reflected in the current design methodology for general purpose processor-based systems, as well as platform-based embedded system design.

## 3. Classics: General purpose, processor-based systems

Most modern processors nowadays rely on advanced microarchitectural mechanisms to improve performance. Typically, during each clock cycle, at least 3-4 instructions are fetched in parallel and then executed in the processor pipeline. To uncover more parallelism, the code is re-scheduled by specialized hardware units that perform register renaming and elimination of dependencies. To this end, we consider a typical superscalar processor configuration as depicted in Fig.4.

The first stage corresponds to accessing the instruction cache (I-cache) and the branch prediction hardware. Sophisticated branch prediction mechanisms are usually implemented in hardware so as to increase performance of modern processors via branch direction speculation. Next, the fetched instruction(s) are decoded and registers are renamed so that the code becomes as free of dependencies as possible. The configuration presented in Fig.4 includes also an instruction window which holds instructions until they are ready to issue and keeps track of existing dependencies. Finally, ready instructions are executed or data cache (D-cache) is accessed and results are written back to the register file for all committed instructions.

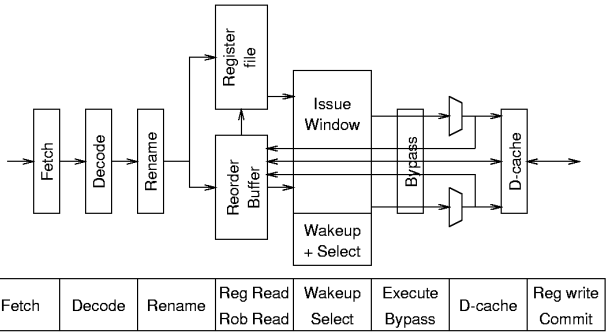



Fig.4 A typical high-end processor pipeline

Most applications running on core processors (either high-end, embedded or application specific) exhibit a wide range of run-time profiles, both within and across applications. This is mainly manifested via non-uniform resource usage, as well as bursty communication patterns among various parts of the pipeline. One such example is the case of an I-cache miss being resolved, or of non-blocking D-cache misses. In the latter case, for example, instructions may proceed normally through the pipeline unless data dependencies are detected. In addition, if there are instructions non-critical to the overall performance (e.g., infrequent floating point operations in integer applications), their execution may proceed at a lower speed without significantly affecting performance.

The above discussion suggests that enough decoupling *exists* among different pipeline stages of a high-end processor so as to allow asynchrony among them either by managing clock skew at run-time, or by simply using multiple clock domains via locally generated clocks.

A possible monolithic, single clock vs. a partitioned, multiple clock domain implementation is shown in Fig.5. In the traditional superscalar out-of-order processor model shown in Figure 5(a), the *instruction flow* consists of fetching instructions from the instruction cache, using the branch predictor for successive fetch addresses. The INT and FP dataflows consist of issuing instructions out of the instruction window to the integer and FP units and forwarding results to dependent instructions. The *memory dataflow* consists of issuing loads to the D-cache and forwarding data to dependent instructions. Introducing high latencies in any of these three crucial flows has an impact on processor's performance and energy consumption.

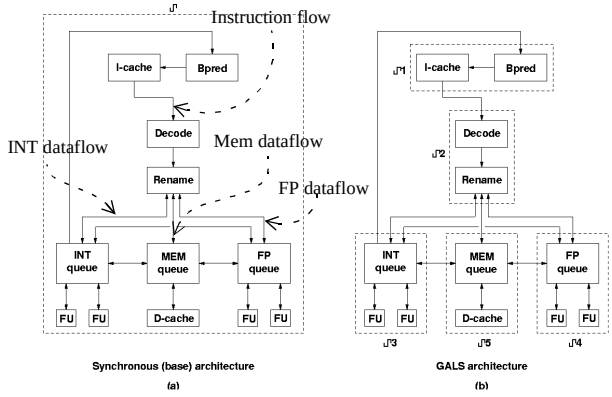



Fig.5 Fully synchronous vs. multiple clock domain cores

However, it is worthwhile to note that detailed analysis of the behavior of both the instruction flow and dataflow suggests a partitioning, at least to some extent. To carry out the analysis, the notion of correlation between two streams can be employed. In general, if two hardware modules exchange information via a buffer, the amount of synchronous communication between them can be characterized by the correlation between the stream produced by one and consumed by the other. This property can be easier characterized via statistical measures applied on the two streams. Specifically, we can employ the *correlation coefficient* [7] of two random variables in order to characterize the potential for decoupled behavior between the two. High values (typically larger than 0.7) indicate high correlation between produced and consumed items, whereas lower values indicate average or low correlation, and thus higher potential for globally asynchronous behavior.

In fact, such an analysis for a typical multimedia application (e.g., MPEG-2 decoder) provides insights into the existing potential for natural decoupling of different domains. For instance, all important computational kernels of the MPEG-2 decoder (Variable-Length Decoder - VLD, Inverse Discrete Cosine Transform - IDCT, Motion Vector Unit - MV) exhibit

very low correlation for the integer dataflow in Fig.5 (correlation coefficient = 0.13 - 0.15) and almost no correlation for the memory dataflow (correlation coefficient = 0.02 - 0.06). The instruction flow shows high correlation (correlation coefficient higher than 0.9), and thus is not a good candidate for decoupling.

Multiple clock domain cores also come with additional potential for power savings [8]. As shown in Fig.6, the total clock power can be reduced by 35% in the case of GALS cores. However, due to increased execution time and higher speculation, the power consumed by the front-end increases slightly, as does the power consumption for the D-cache and execution core. Overall, average power is reduced by 15% for the GALS design. In addition, the speed of these locally synchronous blocks may be gracefully scaled down, while running at a lower voltage, thus providing additional power savings. It can be shown that up to 25% reduction in average power consumption, with less than 15% performance penalty can be achieved by employing fine grain voltage and clock speed scaling. It should be pointed out that there are classes of applications (e.g., FP applications, or applications with small memory footprint) for which fine grain clock and voltage scaling is more efficient than the fully synchronous, voltage scaling scenario (in some cases, more than 15% better than the single clocked system [8]).

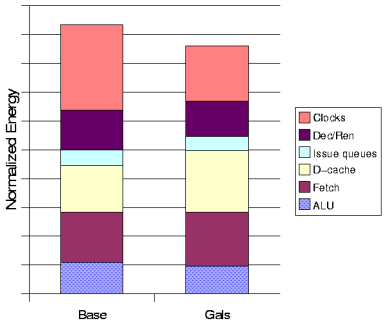



Fig.6 Power consumption breakdown

This analysis shows that $MC^2$ and its incarnation in programmable cores as *GALS microarchitecture* can provide better energy-performance trade-offs than could be discovered otherwise. While many agree that a decentralized microarchitecture, along with use of multiple computation threads on the same core should be the target of future research [3], there is no clear methodology on how the potential for better quality or power-performance operating points can be discovered. $MC^2$ and stream correlation analysis provide a starting point for such methodologies.

## 4. Platform-based design

Platform-based design is the most recent trend in the design methodology of modern embedded systems [9]. A *platform* represents a family of heterogeneous architectures that satisfy a set of architectural constraints imposed to allow re-use of hardware and software components. As such, the success of platform-based design builds upon the ability of enabling design re-use and significantly shortening the time-to-market.

We believe that, again, the $MC^2$ applies to this new design methodology. More precisely, from an application point of view, the key issue is the *automated partitioning* of applications based on system-level communication cost analysis. This is motivated by the observation that, constraining a given application (e.g. MPEG-2) with various input traces (e.g. MPEG-coded video movies with very different scene changes) leads to very different 'clusterings' of the probability distribution that characterize the application itself. Tuning the target architecture to this large spectrum of different probability distributions is the most important development in obtaining efficient mappings with respect to certain performance metrics. This idea is illustrated in [10] which takes a formal approach toward system-level analysis based on Stochastic Automata Networks (SANs). Such a design methodology based on a set of formal techniques for analysis is a critical issue for obtaining high-quality designs since it helps to avoid lengthy profiling simulations for predicting power and performance figures. Considering, for instance, that 5 min. of compressed MPEG-2 video needs roughly 1.2 Gbits of input vectors to simulate, the impact of having such a tool to evaluate power/performance estimates becomes evident.

The entire process graph which corresponds to the MPEG-2 *application* is modelled following the *Producer-Consumer* paradigm (Fig.7). The decoder consists of VLD, IQ/IDCT, MV units, and the associated buffers. To unravel the complete concurrency of processes that describe the application, each process is assumed to have its own space to run so there is no competition for any computing resource. As shown in Fig.7, we have *local* transitions (e.g. between *produce*(*item*) and *wait_buffer* states in the *Producer* process (*VLD)*, as well as *synchronization* transitions between the two automata. During the modeling steps, we model each of these units as processes, and generate their corresponding SANs.

Once we have the SAN model, we can find out its steady-state solution. This can be done by using numerical methods that do *not* require the explicit construction of the global descriptor but can work with the descriptor in its compact form. This is a particular property of SANs which make them very attractive since, due to the special structure resulting from the tensor product, computation is significantly improved using dynamic programming-like techniques. Once the steady-state distribution is known, performance measures such as throughput, utilization, average response time can be easily derived.

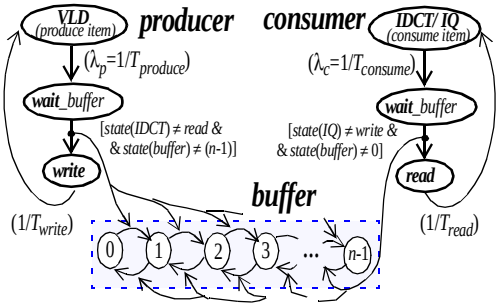



Fig.7 The SAN model of the baseline unit of MPEG-2

The *architecture* modeling step starts with an abstract specification of the platform and produces a SAN model that reflects the behavior of that particular specification. We construct a library of generic blocks that can be combined in a bottom-up fashion to model sophisticated behaviors. The generic building blocks model different types of resources in an architecture, such as processors, communication resources, and memory resources. Defining a complex architecture thus becomes as easy as instantiating building blocks from a library

and interconnecting them. Compared to the laborious work of writing fully functional architecture models (in Verilog/VHDL), this can save a significant amount of time, and therefore enable more exploration of alternative architectures.

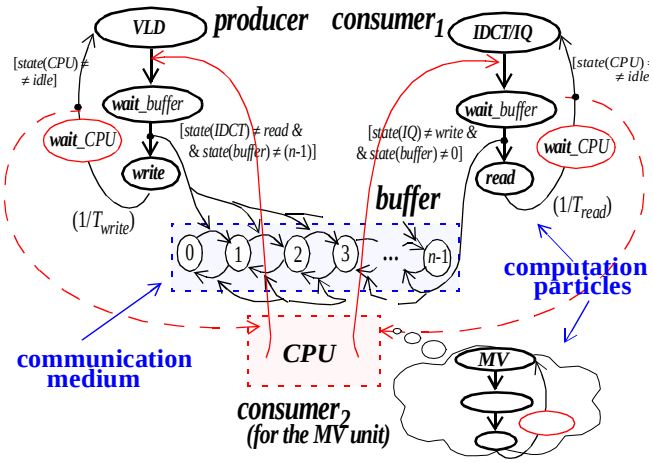



Fig.8 Application mapping onto a single CPU platform

Finally, let assume that we want to decide how to configure a platform which can work in four different ways: one which has three identical CPUs operating at a generic clock frequency $f_0$ (then each process can run on its own processor) and another three architectures where we can use only one physical CPU, but have the freedom of choosing its speed among the values $f_0$, $2f_0$, or $3f_0$. The mapping of our simple *VLD-IDCT/IQ* processes in Fig.7 onto a platform with a single CPU is illustrated in Fig.8. Because these processes[b] have to share now the same CPU, some of the local transitions become synchronizing/functional transitions (e.g. the local transitions with rates $1/T_{produce}$ or $1/T_{consume}$ become synchronized). Moreover, some new states (e.g. *wait_CPU*) have to be introduced to model the new synchronization relationship.

We analyzed in [11] the overall system behavior and presented detailed results under various input traces. Most notably, the average length of buffers associated to the MV and IDCT/IQ units is very different from any worst-case prediction. Based solely on performance figures, the best choice would be a single CPU with speed $3f_0$.

From power dissipation perspective, the breakdown of power-consumption is given in Fig.9. In this figure, Run 1 represents the 'reference' case where the CPU operates at frequency $f_0$; the second and the third runs represent the cases when the CPU speed is $2f_0$ and $3f_0$, respectively. In each case, we indicated the absolute values of power consumption that corresponds to active and idle states of the CPU ($CPU_A$ and $CPU_I$, respectively), as well as the power dissipation due to the MV and IDCT/IQ buffers.

We can see that there is a large variation among the three runs with respect to both the CPU-active power (the $CPU_A$ bar in Fig.9), and the power dissipation in the buffers. Furthermore, we can multiply these power values with the average buffer length predicted by the SAN analysis, and get the *power×delay* characterization of the system. This may be used to quickly decide, for the given set of parameters, which is the best application-architecture combination.

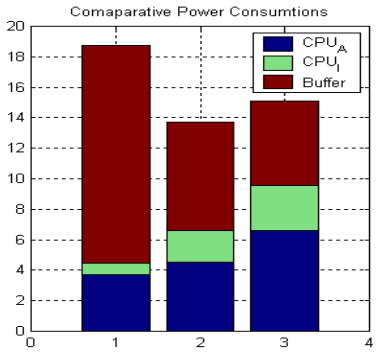



Fig.9 Power-consumption figures for $f_0$, $2f_0$, and $3f_0$

We note that, with respect to the $MC^2$ model, this mapping corresponds to the coalesced resources case illustrated in Fig.3(a). In a more general case, the mapping shown in Fig.8, can be changed to accommodate multiple distributed processing units which will correspond to aggregating processing resources as in Fig.3(b). This will correspond to reaching a "stable state" (e.g. lower *power×delay* product) when computational particles move closer to each other due to, perhaps, the need for low communication overhead.

## 5. Ahead: MC$^2$ in action

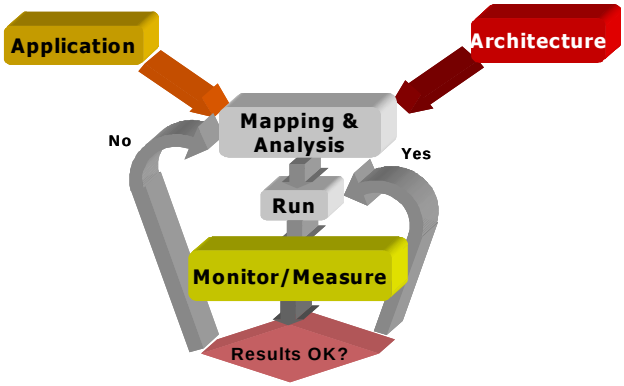
What we have presented so far are two examples where $MC^2$ comes into play via *static aggregation* of computational particles, be it at application, architecture or microarchitecture level. This is because the aggregation is the result of application or architecture partitioning for achieving minimal communication overhead and maintaining quality metrics (such as performance, energy or fault-tolerance) within certain limits. However, we can think about aggregation of computational particles as being done *dynamically* at *run time*. In fact, it is our belief that, for achieving better quality, we need to move the "aggregation" process at run time, by actively re-configuring the architecture and application for better adaptability to the operating environment.

Fig. 10(a) shows the classic design flow used for most of today's electronic systems. The application is mapped onto an architecture or platform (depending upon the type of system) so as to optimize certain metrics of interest and achieve a certain quality for the design. This is basically an iterative process which refines both the application or architecture so as to satisfy certain constraints. The actual mapping is *fixed* and the resources can be shared, for example, in the form of core processors running multiple processes at the same time.

However, in the view of upcoming Systems-on-Chip (SoCs), which are in turn based on specialized on-chip communication networks, a static mapping between application kernels and hardware modules (programmable or not) may *not* exploit all available trade-offs among various metrics of interest (performance, energy, fault-tolerance). We believe that since on-chip communication will most likely entail similar concepts or techniques as real data networks [6], some of the issues encountered will be the same (e.g. fault-tolerance and network congestion management), while others will be specific to the

b. For simplicity, the second consumer process (for the MV unit) was not explicitly represented in this figure.


(a) Classic design cycle




(b) Lifetime cycle of $MC^2$-based systems

Fig.10 Classic vs. $MC^2$ flow

limited resources available on SoCs (e.g. power-performance trade-off and management).

In our vision, some of the decisions related to application mapping and communication have to be moved at run-time, as shown in Fig.10(b). While the initial mapping of the application onto the platform can reflect a particular set of constraints, these may be changed on-the-fly, depending on the operating conditions. We note that this is a more aggressive approach than the on-chip power management of resources (e.g. [12]) which does *not* consider that application mapping can be changed, or communication architecture reconfigured. In addition, our vision is *orthogonal and complementary* to platform-based design by moving the optimization and management of metrics other than power consumption *at run-time*. The mapping and reconfiguration process of the application onto the underlying architecture is achieved via *explicit* mechanisms, as opposed to classic computing systems where mapping and resource management is done via *implicit* mechanisms. In addition, fault and energy modeling and management become *intrinsic* components for achieving certain levels of *quality of results* or *operational longevity*. To this end, we believe that the following issues extremely important for $MC^2$-based systems:

- *Application modeling and partitioning*

  While these are anyway valid issues in system-level design, the possibility of application re-mapping and/or reconfiguration makes them more challenging. Concurrency extraction becomes then the central issue and providing formal support for efficient mapping onto a set of architectural/communication resources is thus a must.

- *Fault modeling and management*

  While it is recognized that on-chip communication is subject to errors due to cross-talk, soft errors or electromagnetic interference, the same holds for computation. At the same time, non-zero failure rates can appear in on-chip communication due to the interplay between limited bandwidth and collision on shared communication resources. It is thus mandatory to include not only fault models for both computational particles and communication medium, but also mechanisms for their management. It is our belief that not only energy, but also fault-tolerance (and consequently, any performance drop due to non-zero failure of computation/communication) should be managed at run-time as any other resource. Energy, fault-tolerance and performance become the driving forces that gear a given configuration towards a stable state, via dynamic aggregation (as described next).

- *Application mapping, remapping, migration, and reconfiguration*

  Both architecture configuration and mapping of the application on the computational particles are done dynamically, at power-up and whenever the system becomes "unstable" due to undesirable changes in the operating environment (such as, high collision rate, non-responsive nodes in the on-chip network, non-balanced on-chip power density). The mapping of computational kernels to the computational particles is also dynamically adaptable, depending on the needs. While application mapping is very much related to load balancing in real data networks [13], the problem of migrating computational *kernels* (not just nodes) from a given *set* of hardware resources to another one is more challenging, given the limited resources available for performing it. At the very least, hardware modules should be able to save their state and send it over the network to a redundant or spare module able to take over the task. The amount of redundancy (coupled with associated regularity at chip level) plays an important part in achieving certain power-performance trade-offs and has interesting implications in design cost and manufacturing.

- *Hierarchical management of (nano)resources*

  We believe that the re-organization and re-mapping should be achieved by using thin middleware clients, sufficiently simple to achieve the required goals without prohibitive overhead. We also note that the dynamic aggregation process can be applied on a tiered architecture, with *local* vs. *global* or hardware vs. software management of resources, including energy and fault-tolerance. While Fig.3b shows a single level in this hierarchy, one can consider *nano*, *micro* and *macro* levels for aggregating, managing and reconfiguring resources (Fig.11). In such cases, fault modeling and management, power conservation and process/application migration or re-mapping have to be approached differently, depending on the level in the hierarchy.

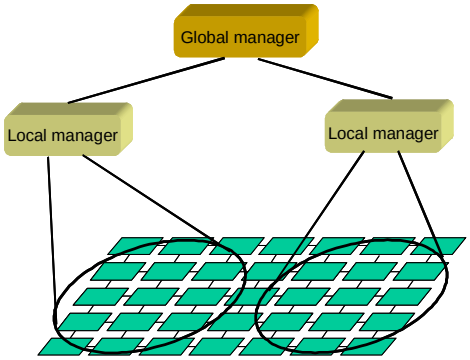



Fig.11 Tiered architecture

## 6. Conclusion

It is generally agreed that the standard ASIC design flow cannot cope with increased levels of integration and complexity of today's systems (hence the need for using GALS design paradigms). It is also recognized that an increasingly important segment of the market is dominated by application specific systems that need tailored design solutions to continue the trend of increasing QoS and power efficiency under given performance constraints. Such applications (especially media processing applications) lend themselves to a "tight local computation/loose global communication" type of modeling. The proposed colloidal model covers both the *software* and *hardware* platform aspects in a unified framework in line with the "tight local computation/loose global communication" paradigm. We believe that such a model will deeply affect how platform designers and IP-core providers think their design flows and products to achieve higher quality and shorter design cycles.

## References

[1] Paul C. Hiemenz, Raj Rajagopalan. Principles of Colloid and Surface Chemistry. Chapters 1 and 13, Marcel Dekker, 3rd Edition, 1992.

[2] W. Maly. IC design in high-cost nanometer-technologies era. In *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, NV, June 2001.

[3] R. Ronen, A. Mendelson, K. Lai, L. Shih-Lien, F. Pollack, and J. P. Shen. Coming Challenges in Architecture and Microarchitecture. *Proc. of the IEEE*, March 2001.

[4] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, NV, June 2001.

[5] L. Carloni and A. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. *Proc. ACM/IEEE Design Automation Conference*, Los Angeles, CA, June 2000.

[6] W. Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. *Proc. ACM/IEEE Design Automation Conference*, Los Angeles, CA, June 2000.

[7] A. Papoulis, Probability, Random Variables, and Stochastic Processes, McGraw--Hill, 1986.

[8] A. Iyer and D. Marculescu. Power Efficiency of Voltage Scaling in Multiple Clock Multiple Voltage Cores. CMU-CSSI Technical Report, 2001.

[9] R.E., Bryant et. al. Limitations and challenges of computer-aided design technology for CMOS VLSI. *Proc. of the IEEE*, Vol.89, No.3, March 2001.

[10] R. Marculescu, A. Nandi. Probabilistic Application Modeling for System-Level Performance Analysis. In *Proc. Design Automation and Test in Europe*, Munich, Germany, March 2001.

[11] A. Nandi, R. Marculescu. System-Level Power/Performance Analysis for Embedded Systems Design. In *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, NV, June 2001.

[12] T. Simunic. Power Management for Netowks on Chip. *Proc. Design Automation and Test in Europe Conference*, Paris, France, Feb. 2002.

[13] A. S. Tanenbaum. Computer Networks. Prentice Hall, 1996.

[14] D. M. Chapiro. Globally Asynchronous Locally Synchronous Systems. Ph.D. thesis, Stanford University, 1984.