(Thesis Abstract)

X- Legion: a Compiler-Approach to Exploit Locality and Portability of Divide-And-Conquer Algorithms

I introduce compiler techniques for the analysis and optimizations of D&C algorithms. I discuss the implementation of these techniques in an *investigative* compiler that I call ***X-Legion*** [1], Fig. 1. The main goal of this approach is the analysis and model of D&C algorithms so to exploit locality and software portability by addressing code optimizations and compiler-driven memory-hierarchy adaptations such as data--cache-line size, data-cache mapping.

[1]The Roman X Legion (the tenth legion) was under the direct order of Julius Caesar during the Gaul's war.

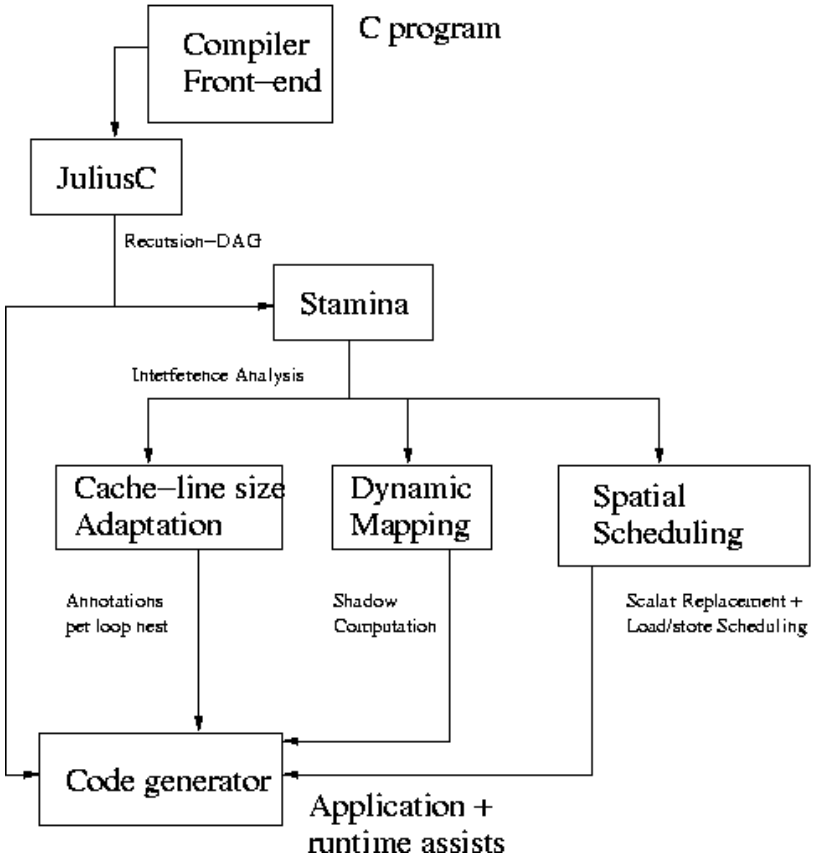

Figure 1: X-Legion Compiler

The D&C input application, written in C, is transformed into an IR, which represents the entire program in a single hierarchical structure. **JuliusC** takes the hierarchical structure and determines the call graph of the program. At this point, JuliusC determines whether a function is *recursive* or a function is a *computation leaf*. In fact, the goal of JuliusC is the analysis of the problem division and, especially, the division of a recursive D&C (implemented in the input application) and the concise representation of such a process by a ***recursion-DAG*** . If this is possible at compile-time, the recursion-DAG can be used for the analysis of the computation leaves and this analysis is performed by **Stamina**. In any case, we can reorganize the application so as we can generate the recursion-DAG at runtime and we use it for the efficient execution of the recursive section/part of the application.

Stamina then extracts the loop nests of the application and analyzes perfect loop nests, which are the most computation demanding (leaf computation). The goal of Stamina is to analyze and quantify the cache interference per memory reference in the inner loop of a loop nest. Such an analysis is used to activate three possible optimizations/adaptations: **cache-line size**, **dynamic mapping** and **spatial scheduling**. In fact, if the leaf computation is afflicted by major cache interference, the leaf computation performance and the application performance can be heavily degraded.

The result of the interference analysis, generated by Stamina, is used to drive the data cache-line size per loop nest introducing a special instruction (or *annotation*). In fact, a large cache-line size reduces the number of data fetches but it may increase cache interference; otherwise a short cache-line size may reduce cache interference, because it also changes the data cache mapping, but it increases the number of data fetches.
An adaptation, which is orthogonal to adaptive cache-line size, is dynamic mapping. In fact, we may tailor the data cache mapping to the application and we make the cache believe that the memory addresses used to store and retrieve data in cache is different from what it is actually used in memory. We call this alternative memory space as **shadow space** and the computation in the new space as **shadow computation.**
At last, we may circumvent the effects of cache interference only by software using a careful memory access scheduling and register allocation. In practice, spatial scheduling is a source-to-source transformation that, using scalar replacement, compacts consecutive in-memory accesses into a sequence of loads into scalar variable (thus into registers); this exploits fully spatial cache locality and reduces the effect of cache interference (without removing interference but reducing the effects).

Description of the basic blocks

JuliusC (**JC**) is an interpreter that generates the recursion-DAG at run-time. It is a proof of concept that the recursion-DAG can be built and the analysis overhead is practical.
Output: JC generates the call graph and the recursion-DAG of the application.
Goals: JC aims at the determination of: the accurate inter-procedural data dependency analysis of the application, the model of the unfolding of recursive algorithms, the characterization of computation leaves and the support of the recursive computation.

Stamina is a tool for the analysis of the effects of cache-line size on cache interference. In fact, given a perfect parameterized loop nest, that is, a perfect loop nests with parameterized loop bounds and memory references with parameterized index, the tool offers a static symbolic analysis and an estimate of the data-cache misses for direct mapped data caches (for different cache-line sizes).
Input: Stamina analyzes a set of (parameterized) perfect loop nests with a static schedule of the memory references (in the inner loop).
Output: Stamina returns a cache-miss estimate as a function of the cache-line size.
Goals: Stamina has been design for the analysis of the loop nests used in the leaf computations of divide-and-conquer algorithms (in particular recursive D&C).

The cache interference may activate three different adaptations
(Hardware only – software-hardware – software only)

Cache-line size adaptation (CLS) is an approach to change (using a special instruction) the cache-line size. Basically, the interference analysis is used to determine the optimal cache-line size per loop nest. Annotations are introduced in the header of a specific loop nest to drive cache-line size at run time.

Input: CLS reads a loop nest and the result of the cache interference analysis.
Output: CLS returns an annotated loop with a special instruction to set the cache-line size at run time.
Goals: CLS is an approach designed for the reduction of the cache-line size when cache interference compromise spatial cache locality so as to decrease overall performance, or otherwise, for the increase of the cache-line size.
Note: this adaptation basically leaves unchanged the computation of the loop nest (in fact, it is a compiler-driven *hardware* adaptation).

Dynamic Mapping (DM) is a software-hardware approach aimed to achieve zero data-cache interference, which is orthogonal to cache-line size adaptation. Using the computation power of the processor we can tailor the data cache mapping so as to circumvent cache interference. We embedded *shadow codes* in the original code and this shadow code will drive the data-cache mapping at runtime.
Input: DM takes as input an application that has affine functions for the representation of the index computations used to access arrays.
Output: DM embeds shadow computations (shadow index computations used in shadow load/store, which drive adaptive data-cache mapping) into the original code.
Goals: DM targets at the support of application-dependent cache mapping however it does not change the hardware organization (i.e., ultimately, it is a direct mapped data cache), so as to nullify cache interference.
Note: In practice, the approach makes believe the cache that different sequences of addresses are accessed during the execution of the application. These changes could be used as feed back for a new cache-interference analysis.

Spatial scheduling (SC) is a cache-interference aware register allocation algorithm. If the application has spatial locality but there is cache interference, we can still exploit locality by storing the contents of an entire cache line in registers and then we perform the computation.
Input: SC considers the body of the inner loop in a loop nest and the result of the cache interference analysis as input.
Output: SC reorganizes the schedule of the load/store and, therefore, the register management.
Goals: SC aims at the reduction of the interference negative effects and exploitation fully of spatial locality using the register file to store a cache line -- all at once -- so as to avoid multiple cache-line fetches due to interference. (But it does not avoid interference.)
Note: this step will change the schedule of load stores. This new schedule could be used as feed back to Stamina and obtain a new cache-miss estimate.

Experimental setup and experimental results available

I need to support by experimental results that the optimizations and adaptations – here presented -- really boost performance. Unfortunately, the performance evaluation of CLS

and DM cannot be measured directly because there is not a *real* architecture supporting such adaptations. I summarize in the following what I have at this point:

- JC: I have hand-coded Recursive Matrix Multiply (RMM), LU-factorization (RLU) and FFT (RFFT) using the recursion-DAG to drive the execution. I have measure of their performance (MFLOPS) for several architectures (and comparison with other scientific libraries).

- CLS: I have cache miss estimates for Matrix Multiply (as leaf computation) and SWIM. I have verified the estimate by simulation. At this time, I do not have a measure of the performance improvements due to cache line size.

- DM: I have hand-coded RMM and RFFT and I have an estimate of the real performance (MFLOPS)

- SC: I have hand-coded the approach for SWIM and I have measure of real performance for three architectures based on Sparc and MIPS processors (and three compilers).

Experimental Results and Setup - Proposed

What I want to do for the thesis is to take RMM, RFFT, SWIM (for a set of inputs) and use a cycle accurate simulator (e.g., based on simplescalar) and evaluate their performance when the optimizations are applied separately and in combination.

Compiler Locality Analysis of Divide-And-Conquer Algorithms To Exploit Adaptive System of Systems (Four-year Research Plan)

With highly configurable SoS (System of Systems) available, composed of in-core processors and several off-core processors supported by hierarchical memories throughout the system and as basic communication means, locality of data and computation is the key to achieve high performance. Intuitively, an algorithm has **locality** when it exploits a decomposition of the computation into sub-computations and each sub-computation has a high ratio of the number of (basic) operations over the number of required inputs; we identify this ratio as the **locality ratio**. In fact, the locality ratio of an algorithm quantifies the algorithm's ability to complete each sub-computation independently (e.g., on a specific FPGA) and to hide communication latency (e.g., to/from a shared data cache). In practice, an algorithm with high locality will use locally available data efficiently, striving to accomplish the minimum number of off-cache communications for the largest number of operations. Also, an algorithm will exploit independent sub-computations and will reduce communications and communication overhead among dependent sub-computations. In fact, from a performance prospective point of view, locality translates into an efficient use of system resources such as memory hierarchies and spatial parallelism, now available in multi-core processors, naturally and, in turn, this is the key to high performance.

We have demonstrated for high-performance systems how different algorithms – even for the same problem - have extremely different locality, and even worse, different implementations of the same algorithm may have largely different locality and performance. For the reconfigurable computing system, especially for different-in-kind applications, this will imply an underutilization of the (expensive) system, for which we may find an optimal configuration if we look for it hard enough.

The first step and conceptually the most important, towards the efficient utilization of SoS and reconfigurable systems lies in the choice of the algorithm and in the way an algorithm is formulated. Many important scientific computations (of interest to numerous applications) are intuitively expressed in recursive form; some important examples are: Fourier Transform - FFT, matrix factorization – LU-factorization, kernel for the solution of linear systems of equations engines. Recursive algorithms exploit data locality and naturally yield parallel implementations.

Despite this, scientific computations have not traditionally been implemented in recursive form, even though these algorithms are often initially described in books or papers in recursive, or blocked format, only to be later translated into non-recursive form. This was largely due to the perceived inadequacies of existing compilation techniques/tools and the resulting inefficient execution of recursive programs. In practice, the organization and description of an algorithm by careful hand tuning may lead to an efficient analysis and implementation exploiting locality and parallelism.

In fact, locality is a property of the algorithm, or stated differently in the context of static analysis tools, locality is a property available at the most abstract level – i.e., at algorithm level. The ability to recognize locality and allow the final executable to exploit such locality is possible only if the *semantics* are recognized and *maintained* as long as necessary, even at the lowest level of representation of the algorithm (i.e., machine code). Locality may steer compiler-driven optimizations involving code reorganization and, more interestingly, it may steer architecture optimizations or core reconfigurations or memory hierarchy adaptations to optimize overall performance: hierarchical decomposition locality is useful as well in matching overall resources adaptation to the problem at hand.

While we may assess the algorithm's properties at different points in time during the algorithm compilation-execution life time, there are also different types of (locality) abstractions presentable/available all along. Actually, the choice of a specific semantic abstraction is a function of its application (i.e., when/how it is used) and how it matches existing hardware or to drive required adaptations. Intuitively, to reduce the dollar/FLOP ratio with a given underlying hardware, semantic information may be used – at compile time – to improve the initial program as well as – at run time – to drive hardware configuration/networking of several, and spatially close, hardware components (e.g., FPGAs). The ability of a static-analysis tool to generate and retain semantic properties assure the ability to generate an executable able to exploit the inherent properties of the initial algorithm, even the ones expressed at algorithm level; that is, information about the algorithm as a whole.

At the beginning, we will target at the optimization of Divide-and-Conquer algorithms because they are intuitive, portable and efficient top-down solutions for problems on large volumes of data as many of those relevant to H&RT; we will then target more challenging examples of scientific computation. D&C algorithms are ideal because they exploit locality of the resources naturally and they decompose the computation in sub-computation explicitly. These features allow experienced designers (or very specific automatic tools) to tailor rather quickly and naturally algorithms to different uniprocessor systems as well as to multiprocessor systems. Unfortunately, the problem decomposition is based on the input sizes and on control-flow statements – i.e., recursive function calls and these complicate the analysis of the potential performance of the applications.

We propose a compile-time approach that empowers the application so to determine a computational model on the fly (and with no profiling) at run time. The computational

model describes how the computation unfolds in time and it offers the opportunity to the algorithm itself, or key features of the underlying architecture, to adapt on demand. Using a simplified data-dependency analysis, a static analysis tool can annotate part of the application responsible to the decomposition process. These annotations are used at run time to unfold the computation of a specific recursive function definition and create a Directed Acyclic Graph (DAG), that we call **recursion-DAG**. This graph can be used to quantify how many times a function is called, what is the space complexity for a particular function call or a family of function calls, what is the data dependency among function calls. This information can be used to specialize a very frequent function call at run time, to map function calls to processors, to quantify whether or not the decomposition will be effective, and to demand reconfiguration (as needed) to optimize overall applications/systems performance.

This information is extremely beneficial for the application of compiler-driven optimizations for rather complex computational algorithms on adaptive and complex SoS, allowing the software to interact with the hardware in search for the best configurations to optimize overall performance. The ultimate goal for any high-performance systems is the realization of a self-interactive and efficient bind among software and hardware so as to achieve the best performance and, in turn, to increase the life-time for both components because of ease of reuse-ability. This makes our approach cost effective and suitable for long-lived SoS. Furthermore, this synergy of optimizations will permit reliable, predictable and easy to reproduce performance, shortening the test and verification of ground-based computing (even for a rather complex combination of mixed FPGAs and SoS).

In addition, the recursion-DAG computational model provides a natural framework for debugging and interactive parallel code development. As part of the proposal, we will also develop such a tool, which should greatly facilitate the development of widely parallel code within our system thereby hopefully enhancing and speeding up acceptance of the tool-set within the scientific community. In fact, the debugging process can start even before the execution of the hardware development/configuration, reducing the delivery time to on-board systems, decoupling the debugging process from the actual execution and therefore improving the utilization of the ground computing system. Furthermore, this computation model can be extremely useful because it can be used to emulate efficiently the complex interaction among software and underlying hardware composed of FPGA and reconfigurable SoS.

List of tasks

- **Partitioning**

  Partitioning applications into manageable-size computations to be organized as a hierarchical structure to exploit parallelism at a high level, allowing the

communication and coordination among parallel applications spawned on-the-fly from different source processes to satisfy requirements on-demand.

- **Intuitive user interface**

  Ultimately, the goal is to offer a friendly user interface, comparable to (at least) the one used by general purpose compiler, to introduce a lite-weigh and reconfigurable software environment for SoS able to analyze/generate applications aimed at the full utilization of reconfigurable, highly parallel reconfigurable SoS.

- **Compilation**
  We envision the following static analysis organization among the three different phases of the compilation-execution process as follows:

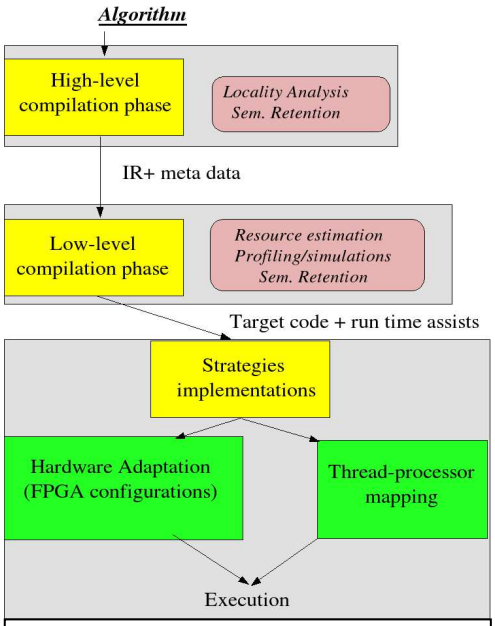

Figure 2: Compilation and execution process

- **High-level Locality-Analysis phase.**

This phase of the analysis is completely oblivious of the underlying computational architecture and it turns its attention on the property of the application/input under investigation. During this phase, we propose a **static-locality analysis** of the input program, which generates an intermediate representation at a function level. We base our analysis on **inter-procedural analyses** to capture the relation among function calls and data among function calls. **Static data dependency** (**intra** and **inter-procedural data dependency analysis**) among operands of function calls are determined so as to specify at this time data dependent and data independent function calls. The relation among function calls is summarized by annotations on the formal parameters in function definitions. These, in turn, will be used in the following phases to extract information about the unfolding of the computation. The result of the analysis is a **tree-based IR** and the **data dependency graph**, the result of the locality analyses are presented as annotations (e.g., comments).

- **Resource-estimation analysis**
  This phase of the analysis aims at the analysis of the computational needs of the application; that is, specific analyses, aiming to capture these application's needs

about resources, are preformed at this time. **Basic-block processor independent optimizations** are performed during this phase to trim superfluous code and operations. We then propose a **resource estimation analysis** based on **cache behavior analysis** through **static analysis**, **profiling** or **simulations** for specific basic blocks as well as for functions. For example, static analysis to determine the cache behavior for basic loop nests may be performed so as to determine whether any cache adaptations is beneficial. As result of this phase, **specialized routines** may be introduced in the original input program as **run-time assists**. These assists will be responsible to interact with the architecture/Operating system so as to activate application-driven adaptation/optimizations.

- **Run-time**
  The executable is executed with the run-time assists. The execution of the algorithms is driven by the assist execution in particular:
  - **Assists Generation**
    At run time, the assists generate a concise representation of the computation ahead, a DAG, **recursion-DAG**

  - **Strategy implementation**
    The recursion-DAG is used for
    - the adaptation of the application's so as to maximize the utilization of the architecture,
    - parallelism extraction,
    - estimation of communication overhead,

  - **Thread-processor mapping**
    Using the strategy implementations and the recursion-DAG, we propose an approach for the investigation, instantiation and mapping of threads and processors.

  - **Hardware adaptations**
    Application-driven adaptations/optimizations of the architectures are fired up so as the architecture changes specific features dynamically to fully satisfy the application's need.

    - Configuration of FPGAs
    - Configuration of caches as means for communication among different threads/processors as well as hierarchical processor allocation as arbiter/distributor of common data among different SoS or specific FPGA.

Time Schedule

A first tentative schedule of the project development will apply a cyclic water-fall paradigm with iterative-successive refinements. At the end of each phase, we target at the design of a prototype, proof-of-concept, bridging together a practical implementations and semantic abstractions to exploit performance. We envision the plan of the project distributed in four year (Y1, Y2, Y3, and Y4) and each year is composed of four quarters (Q1, Q2, Q3, and Q4.

We envision three phases:

1. **Locality analysis phase**
   System-oblivious framework design (Y1Q1-Y2Q4)

   - Algorithms organization, analysis and formulation (Y1Q1-Q3)
     - Statement of the language features/characteristics
     - Statement of hardware/simulation requirements

   - High-level compilation phase (Y1Q2-Y2Q3)
     - Specialized Inter-procedural dependency analysis (Y1Q2-Q4)
     - Semantic retention of call-graph information (Y1Q3-Y2Q1)
       - Advanced inter-procedural data dependency
       - Locality and parallelism formulation at algorithm-level.
     - Locality analysis and semantic representation (Y1Q2-Y2Q2)
     - Closure property verification/inspection (Y2Q2-Q3)

   - Design of the Locality assists generators (Y2Q1-Y2Q4)
     - Recursion-DAG generation
     - What is going to *maintain* and *exploit* locality at run-time

   - Integration and *ad-hoc* test/verification (Y2Q4)
     - Benchmarking

   - High level debugging framework and visualization (Y1Q3-Y2Q3)
     - Debugging tools for the algorithms at hand and for the framework itself

2. **Resource-Estimation phase**
   Resource estimation, processor oblivious frame work design (Y2-Y3)

   - Design and prototyping of techniques for specific resource representation and collection (concise/internal representation of SoS (Y2Q1-Q2)

- Resource-estimation analysis (Y2Q2 – Y3Q4)
  - High-level analysis semantics retention application (Y2Q2 – Y3Q2) to quantify/estimate (from the application point of view):
    - Number of processors/threads
    - Communications requirements
      - Off-cache communications
      - Off-cluster communications
      - Main memory space requirements
    - Hot-spots determination (most called functions-basic blocks)
  - Profiling-based analysis (from the hardware system point of view) (Y3Q1-Q4)
    - Cache adaptation simulation
    - Processor-simulation profiling

- Design of the resource allocation/representation assists generators (Y2Q3-Q4)
  - What is going to probe the architecture and map application specific needs to specific hardware

- Integration and *ad-hoc* test/verification (Y3Q4-Y4Q1)
  - Benchmarking

- Low-level debugging framework and visualization (Y3Q2-Y4Q1)
  - Debugging tools for the estimation tools and for the framework itself

3. Run-time Phase

Full scale integration (Y3-Y4)

- Design of the run-time framework for the execution and coordination of application and assists (Y3Q1-Y4Q2)

- Techniques design and prototyping for run-time optimizations (Y3Q2-Y4Q2)
  - Processes/threads mapping to single processors, to SoS , to cluster of FPGAs
  - Threads clusterings
  - System configurations and adaptations

- Integration and *ad-hoc* test/verification (Y4Q2-Y4Q4)
  - Benchmarking

- Run-time debugging framework and visualization (Y4Q1-Y4Q4)
  - Debugging tools for the estimation tools and for the framework itself