STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 1, 2013

# TOWARDS A REGION-BASED CALCULUS FOR ENERGY-AWARE PROGRAMMING

FLORIN CRĂCIUN, SIMONA MOTOGNA, AND BAZIL PÂRV

Abstract. Energy efficiency has become one of the most critical software metric both for cloud computing servers as well as for mobile phones, ipads or sensor networks. Much of the research on the reducing energy consumption has been focused on low-power architectures, operating systems and compiler optimizations. Recent studies have been started to explore how programming language technologies can help reason about energy management. In this context our paper discusses how our region calculus used before to manage the heap memory can be adapted to control the energy consumption.

## 1. Introduction

In the recent decade, energy-aware computing systems (e.g. mobile devices, wireless sensor nodes, cloud computing servers) had a rapid evolution. The transformation of mobile devices (especially smartphones and iPads) into general-purpose computing platforms had an important impact on considering the energy as a first class design constraint for many software applications. Saving energy can extend the battery lifetime and increase the mobility or can reduce the maintenance costs of data-centers.

Much of the research on energy management has been focused on the optimizations for the *energy-aware execution* of the software programs. The optimization techniques (see [KM08] for a survey) have been developed at different layers of the compute stack (e.g. digital circuits, architecture, operating systems and compilers). They are mainly dynamic approaches based on online monitoring or offline profiling.

Recent studies have been started to explore how programming language technologies can help reason about energy management. A new paradigm *energy-aware programming* has been proposed in order to aid developers to

Received by the editors: March 16, 2013.

2010 *Mathematics Subject Classification.* 68Q60, 68N30.

1998 *CR Categories and Descriptors.* D.2.4 [**Software**]: Software engineering – *Software/Program Verification.*

*Key words and phrases.* energy-aware programming, region calculus, program analysis.

write energy-efficient programs in the first place. Exposing energy considerations at the programming language level can enable a new set of energy optimizations and can allow the program to have a direct control of the energy management techniques from the lower layers of the compute stack.

This paper analyses the new paradigm approaches and discusses a possible unification of them under a general region calculus for energy consumption control. Section 2 introduces the concepts of our previous work on using region calculus for memory management. Section 3 presents the challenges of the new paradigm. Section 4 discusses our proposal while Section 5 concludes the paper.

## 2. Region Calculus for Memory Management

Region types have been introduced to manage the heap memory at compile time. Region-based memory management systems allocate each new object into a program specified region, with the entire set of objects in each region deallocated simultaneously when the region is deleted. The first safe region-based memory system has been developed by Tofte and Talpin [TT94, TT97] for a functional language. Later, several projects have investigated the use of region-based memory management for C-like languages (e.g. Cyclone [GMJ$^+$02]) and object-oriented languages [BSBR03].

In our previous work [Cra08, CCQR04, CQC08, SCC08], we have developed an automatic region type inference system for object-oriented paradigm. Our compiler automatically augments unannotated object-oriented programs with regions type declarations and inserts region allocation/deallocation instructions that achieve a safe memory management. Our work uses lexically-scoped regions such that the memory is organised as a stack of regions. Regions are memory blocks that are allocated and deallocated by the construct *letreg r in e*, where the region *r* can only be used to allocate objects in the program *e*. All objects allocated into a region have the same lifetime. Dangling references are a safety issue for region-based memory management. Our work allows only non-dangling references which originate from objects placed in a younger region and point to objects placed either in an older region or inside the same region. Relations between regions and non-dangling references conditions are expressed as lifetime constraints between objects regions.

Recently, region assertions have also been used to control the possible aliasing [ABB06] in information flow analyses. A new regional logic [BNR08, RBN12] has been proposed to reason about mutation and separation, via variables of type region (finite sets of object references).

## 3. Energy-Aware Programming Paradigm

Energy consumption is a combined effect of many hardware components (such as CPUs, caches, DRAMs, I/O devices) which interact in complex ways. Therefore energy consumption control is a challenging task for programmers. Energy-aware programming paradigm proposes different programming models and logical frameworks that can help developers to reason about energy consumption. However developers are assumed to have minimal knowledge about energy consumption. In general the new proposed programming models assure an efficient and correct control of the hardware-level energy management through special programming language constructions (e.g. special annotations for types or special instructions). Analysing the recent approaches proposed for energy-aware programming we have identified the following main directions: programming using controlled approximations [SDF$^+$11, LPMZ11, BC10, CKMR12], programming using phased behaviours [CZSL12], and programming according to the battery energy states [CZSL12, SKG$^+$07].

3.1. **Programming using Controlled Approximations.** The key observation of this programming model is that the programs spend a significant amount of energy guaranteeing correctness. However the programs have portions that are more resilient to errors and portions that are critical and must be protected from errors. Therefore non-critical portions of the programs can save a significant amount of energy by using approximate computations. Approximate computations might consist of approximate storage (e.g. reducing refresh power in DRAM memories [LPMZ11], unreliable registers and data caches), approximate operations (e.g. instructions for approximate integer ALU operations as well as approximate floating point operations) and algorithmic approximation (e.g. approximation of the expensive functions and loops [BC10]).

Distinguishing between the critical and non-critical portions of a program is the main challenging task of this programming model. EnerJ [SDF$^+$11] proposes a type system that isolates the precise portion of the program from the approximate portion. That means it prevents a direct flow of data from approximate to precise variables. It also allows programmers to compose programs from approximate and precise components safely. Later a more complete architectural support for approximate programming has been developed in [ESCB12]. Recently, a relational assertion logic [CKMR12] has been proposed to express and verify the properties of program approximations.

3.2. **Programming using Phased Behaviours.** The key observation of this programming model is that different program fragments have distinct patterns of CPU usage, memory accesses, cache misses, and I/O operations which lead to a distinct pattern of energy consumption. Therefore a program usually have phased behaviours of energy consumption. The rate of energy consumption is steady within a phase but different across.

The main challenging task of this programming model is to determine the number of phases and the boundary of each of them. Energy types [CZSL12] allow the programmer to specify phased behaviours by using phase type annotations. The type system can enforce the phase distinction (each data and operation must commit to only one phase) and the phase isolation (any cross-phase interaction can be done only with type coercion). Phase type information can control the CPU dynamic voltage and frequency scaling (DVFS). DVFS is based on the observation that is most advantageous to scale down the CPU frequency (such that energy can be saved) when the CPU is least busy (such that the performance is the least affected). The challenging task for applying DVFS is to choose the right scaling point and the right scaling factor. In this case the solution reduces to finding the appropriate boundaries for the phases.

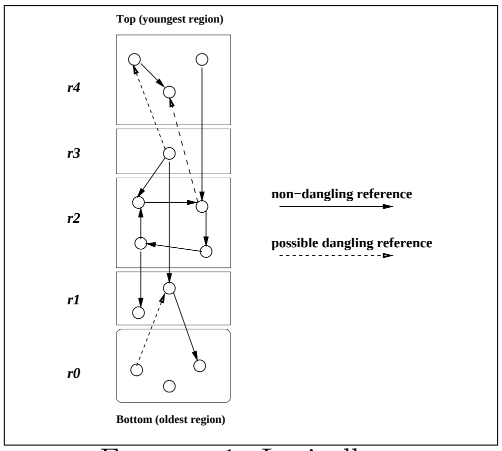

FIGURE 1. Lexically-Scoped Regions

3.3. **Programming according to the Battery Energy States.** The key observation of this programming model is that the different choices to implement an application may consume different levels of energy and be best used in different battery energy states.

The main challenging task of this programming model is to adapt the program to the available energy. Energy types [CZSL12] uses mode type annotations to indicate the expected energy usage context associated with specific data or operations. In [SKG$^+$07] a coordination language is proposed in order to dynamically adapt the system to available energy.

## 4. A Unified Region Calculus for Energy Management

In this section we propose a general region calculus for both data and code that can unify all three energy-aware programming models presented in Section 3. This proposal extends our region calculus used before to manage the heap memory at compile time [Cra08]. First we illustrate our previous region calculus by a simple Java code example and then we analyse the modifications of our calculus to support energy-aware programming models.

Our region calculus uses lexically-scoped regions such that the heap memory is organised as a stack of regions, as illustrated in Figure 1. Regions are memory blocks that are allocated and deallocated by the construct *letreg r in e*, where the region *r* can only be used to allocate objects in the program *e*. The older regions (with longer lifetime) are allocated at the bottom of the stack while the younger regions (with shorter lifetime) are at the top.

Dangling references are a safety issue for region-based memory management. Figure 1 shows two kinds of references: non-dangling references and possible dangling references. Non-dangling references originate from objects placed in a younger region and point to objects placed either in an older region or inside the same region. Possible dangling references occur when objects placed in an older region point to objects placed in a younger region. They turn into dangling references when the younger region is deallocated. Using a dangling reference to access memory is unsafe because the accessed memory may have been recycled to store other objects. There are two approaches to eliminating this problem. The first approach allows the program to create dangling references, but uses an effect-based region type system to ensure that the program never accesses memory through a dangling reference. The second approach uses a region type system to prevent the program from creating dangling references at all. Our work has adopted the second approach. Let us consider the example of Figure 2, the Pair object is allocated in region *r4* which is the top of the regions stack. The two fields of the Pair *p* are allocated in two regions *ra* and *rb* which must be older than or the same as *r4*. All the safety requirements are guaranteed by our type system [Cra08]. In addition we have developed the first automatic region type inference system for object-oriented paradigm [Cra08]. Our compiler automatically augments unannotated object-oriented programs with regions type declarations and inserts region allocation/deallocation instructions that achieve a safe memory management.

```
letreg r4 in {
  Pair p;
  Object a,b;
  ...
  a = new Object<ra>();
  b = new Object<rb>();
  p = new Pair<r4>(a,b);
  ...
}
```

FIGURE 2. Memory Regions Example

In this proposal we assume that the regions for energy management are manually introduced by the programmers while our energy type systems guarantees the appropriate safety conditions for using energy regions.

4.1. **Programming using Controlled Approximations.** In order to distinguish between the critical and non-critical fragments (both code and data) of the programs we use two kinds of regions: *approximate regions* and *precise regions*. By default, when no region is explicitly given, the code and the data are in a precise region. Therefore the programmers have to introduce the region programming constructions only for the approximate data and computations. In general an approximate region unifies approximate data storage, approximate computation and approximate algorithms.

Let us consider the example from Figure 3 where the region *rApprox* denotes an approximate program fragment. Therefore all the variables declared inside this region (e.g. *a*, *c*), all the memory allocations done inside this region (e.g. *b*), and all the operators computations done inside this region must be approximate. In the case of a function call that is executed inside of an approximate region (e.g. *f(y)*, its computation can be done according to the regions of the function body. However the function call result is stored in an approximate variable ($x$ in our example). Our model is portable, the compiler is entirely responsible for choosing the energy-saving mechanisms for approximate data and computations from an approximate region. The safety requirement that must be guaranteed by our type system is that the approximate data cannot affect precise data. However it is important that data be occasionally allowed to break the strict separation enforced by the type system. Therefore our region model provides a special construction that allows the programmers to control explicitly when approximate data can affect precise state.

```
letreg rApprox in {
  int a,c;
  Object b;
  ...
  b = new Object();
  a = c+a;
  x = f(y);
  ...
}
```

FIGURE 3. Approximate Regions Example

4.2. **Programming using Phased Behaviours.** In this energy programming model our energy regions denote the energy states of the different hardware components whose energy consumption contributes to the energy consumption of the programs. In this proposal we restrict our region calculus to

CPU states such that our energy regions represent the CPU frequencies. For example we can have the regions *rH*, *rM*, and *rL* to denote three different frequencies for a CPU: high, medium and low respectively. The programmers can choose different regions for different program fragments execution according to the characteristics of those program fragments.

Let us consider the example from Figure 4 where the region *rH* is used for CPU intensive operations while *rL* is used for I/O operations. Our *letreg* construction corresponds to two instructions: the first which set the CPU frequency at the beginning of the block and the second which restore the previous CPU frequency at the end of the block. The safety requirement for this model refers to the proper usage of the CPU frequencies. A proper usage of CPU frequencies means to reduce the energy consumption without significantly affecting the execution time. It is very difficult to statically guarantee this safety requirement.

```
letreg rH in {
  c=1;
  while (c < 10000){
        ...
        //CPU intensive operations
        ...
  }
  ...
  letreg rL in {
     ...
     //I/O operations
     ...
  }
  c=1;
  while (c < 10000){
        ...
        //CPU intensive operations
        ...
  }
  ...
}
```

FIGURE 4. CPU Frequency Regions Example

## 4.3. **Programming according to the Battery Energy States.**

In this model the energy regions correspond to the battery energy states. Since the battery state cannot be known at compile time these regions are runtime regions. However we can use a special construction (implemented in a special library) that can check the battery state in order to introduce the regions at the compile time. An illustrative example is given in Figure 5. The programmers can choose the code that will be executed according to the battery status. The safety requirement here is to not allow a transition from a low status region to a high status region without an explicit check/change of the battery status.

## 5. Concluding Remarks and Future Work

```
if (batteryState()==High_State){
  letreg rH in {
    ...
  }
}else {
  letreg rL in {
    ...
  }
}
```

Figure 5. Battery State Regions Example

In this paper we analised the new energy-aware programming paradigm, we identified the programming models of the new paradigm and we discussed the challenges of these models. As a solution we proposed a general unified region calculus based on our previous work on region-based memory management. Our intention is to formalize the proposed calculus and to develop an energy type system that can enforce the energy safety requirements for all the programming models of the new paradigm. We also like to develop a compile-time energy evaluation model that can guide the programmers to find the appropriate places for energy regions in a program. Our work is a small step towards designing programming models for energy efficiency.

## Acknowledgment

This work was possible with the financial support of the Sectoral Operational Program for Human Resources Development 2007-2013, co-financed by the European Social Fund, within the project POSDRU 89/1.5/S/60189 with the title Postdoctoral Programs for Sustainable Development in a Knowledge Based Society.

## References

[ABB06] Amtoft, T., Bandhakavi, S., and Banerjee, A. A logic for information flow in object-oriented programs. In *POPL*, pages 91–102. ACM, 2006.

[BC10] Baek, W. and Chilimbi, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209. ACM, 2010.

[BNR08] Banerjee, A., Naumann, D. A., and Rosenberg, S. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411. 2008.

[BSBR03] Boyapati, C., Salcianu, A., Beebee, W., Jr., and Rinard, M. Ownership types for safe region-based memory management in real-time java. In *PLDI*, pages 324–337. ACM, 2003.

[CCQR04] Chin, W.-N., Craciun, F., Qin, S., and Rinard, M. C. Region inference for an object-oriented language. In *PLDI*, pages 243–254. ACM, 2004.

[CKMR12] Carbin, M., Kim, D., Misailovic, S., and Rinard, M. C. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, pages 169–180. ACM, 2012.

[CQC08] Craciun, F., Qin, S., and Chin, W.-N. A formal soundness proof of region-based memory management for object-oriented paradigm. In *ICFEM*, pages 126–146. 2008.

[Cra08] Craciun, F. *Advanced Type Systems for Object-Oriented Languages.* PhD Thesis, National University of Singapore, 2008.

[CZSL12] Cohen, M., Zhu, H. S., Senem, E. E., and Liu, Y. D. Energy types. In *OOPSLA*, pages 831–850. ACM, 2012.

[ESCB12] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Architecture support for disciplined approximate programming. In *ASPLOS*, pages 301–312. ACM, 2012.

[GMJ$^+$02] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based memory management in cyclone. In *PLDI*, pages 282–293. ACM, 2002.

[KM08] Kaxiras, S. and Martonosi, M. *Computer Architecture Techniques for Power-Efficiency.* Morgan and Claypool Publishers, 1st edition, 2008.

[LPMZ11] Liu, S., Pattabiraman, K., Moscibroda, T., and Zorn, B. G. Flikker: saving dram refresh-power through critical data partitioning. In *ASPLOS*, pages 213–224. ACM, 2011. ISBN 978-1-4503-0266-1.

[RBN12] Rosenberg, S., Banerjee, A., and Naumann, D. A. Decision procedures for region logic. In *VMCAI*, pages 379–395. 2012.

[SCC08] Stefan, A., Craciun, F., and Chin, W.-N. A flow-sensitive region inference for cli. In *APLAS*, pages 19–35. 2008.

[SDF$^+$11] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. Enerj: approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174. ACM, 2011.

[SKG$^+$07] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. Eon: a language and runtime system for perpetual systems. In *SenSys*, pages 161–174. ACM, 2007.

[TT94] Tofte, M. and Talpin, J.-P. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL*, pages 188–201. 1994.

[TT97] Tofte, M. and Talpin, J.-P. Region-based memory management. *Inf. Comput.*, 132(1997)(2):109–176.

Department of Computer Science, Babeş-Bolyai University, 1 M. Kogălniceanu St., 400084 Cluj-Napoca, Romania

*E-mail address*: `craciunf@cs.ubbcluj.ro,motogna@cs.ubbcluj.ro,bparv@cs.ubbcluj.ro`