# Programming Strategies for Contextual Runtime Specialization

Tiago Carvalho, Faculdade de Engenharia da Universidade do Porto
Pedro Pinto, Faculdade de Engenharia da Universidade do Porto
João M. P. Cardoso, Faculdade de Engenharia da Universidade do Porto

Runtime adaptability is expected to adjust the application and the mapping of computations according to usage contexts, operating environments, resources availability, etc. One of the runtime adaptability possibilities is the use of specialized code according to data workloads and environments. Traditional approaches use multiple code versions generated offline and, during runtime, a strategy is responsible to select a code version. Moving to runtime code generation can achieve important improvements but may impose unacceptable overhead. This paper presents an aspect-oriented programming approach for runtime adaptability. We focus on a separation of concerns (strategies vs. application) promoted by a domain-specific language for programming runtime strategies. Our strategies allow runtime specialization based on contextual information. We demonstrate our approach with examples from image processing, which depict the benefits of runtime specialization and illustrate how several factors need to be considered to efficiently adapt the application.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

Additional Key Words and Phrases: Runtime Adaptability, Program Specialization, Aspect-Oriented Programming, Domain-Specific Languages

**ACM Reference Format:**
Int. Workshop on Dynamic Compilation V, N, Article A (January 2015), 10 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

## 1. INTRODUCTION

In future, advanced, embedded computing systems, especially the ones in highly dynamic environments, applications may have to adapt to changes in contextual information (e.g., users location and activity), or to changes in resource availability (e.g., energy [Flinn and Satyanarayanan 2004] and connectivity) [Mukhija and Glinz 2005]. Adaptability may include changes in application parameters, application functionalities, selection among different algorithms (e.g., differing in computational complexity), different compiler optimizations, hardware/software partitioning schemes [Stitt et al. 2003], options to map computations to reconfigurable units [Vahid et al. 2008], management of system resources (e.g., switching and/or deactivating sensors), etc.

One of the runtime adaptability possibilities is the use of specialized code according to data workloads and environments. Traditional approaches use multiple code versions generated offline by a compiler and during runtime an adaptive strategy is responsible to select among the code versions [Diniz and Rinard 1997]. Moving compiler optimizations and code generation to runtime can achieve important improvements [Kistler 1997] but may impose unacceptable overhead and in order to keep the overhead low the compiler analysis and optimizations might be circumscribed to specific and low-overhead optimizations (as is the example of the Just-In-Time (JIT) compilers).

We present a template-based Java bytecode generation approach to reduce the runtime overhead while making possible to generate specialized versions of certain codes. The main idea is to substitute the multiple code versions produced by an offline compiler for a set of computations by a code generator based on a template. Our first examples of template-based code generators are presented in this paper and are used

This work is supported by the Fundação para a Ciência e a Tecnologia, under PhD grant SFRH/BD/90507/2012.

in the context of an odd-even sorting network in the examples of median and sort and of a Sobel image processing operator.

This work is part of our ongoing research for novel techniques to both program and map runtime adaptability strategies, in the context of advanced embedded computing systems. The specification of adaptation strategies is exposed to developers as a high-level domain-specific language (DSL), based on LARA language [Cardoso et al. 2012]. An approach promoting the separation of concerns is beneficial as it makes easier the debug, verification, optimization, specialization, and the mapping of the behavior responsible for the runtime adaptation to specific computing cores. Furthermore, a separation of concerns may require only minor changes to the original application.

This paper is organized as follows. Section 2 shows a motivational example for the ongoing work and some preliminary results. Section 3 describes our approach in providing program runtime adaptability. Section 4 presents some experimental results. Section 5 shows the most relevant related work on runtime adaptability. Finally, Section 6 draws some conclusions and describes future work.

## 2. MOTIVATIONAL EXAMPLE

Software applications are usually developed in a generic fashion, in a way that different input data, value ranges, execution parameters, system parameters, etc., may be considered by reusing the same code. Most contextual information is only available at runtime (e.g., input data, program and system configuration) and if used can provide an advantageous position to improve the efficiency of an application. For instance, certain input ranges allow specialized versions of a program to achieve considerably higher performance than the original (generic) version. The specialized version should be chosen for execution, providing that it has better performance and produces the same output (or accurately similar) as the original version.

Having different algorithms, the selection of the best implementation, within a set of values, is dependent on different parameters, such as number of values, the values range, or the data structure used to store the values. Some of these parameters are only available at runtime, making the specialization, with these parameters, impracticable at compile-time.

As an example, consider the calculation of the median of a number of values. This calculation is commonly used in different areas, such as in statistics and image processing. The most common median approach retrieves the middle value of a list sorted with algorithms such as quicksort, counting sort, and sorting networks [Cormen et al. 2001]. To select the median value, it is not required to completely sort all values, as only the median value has to be in the correct position. The sorting networks are practicable examples, in which specialized versions, focused on retrieving the median value, can provide better performance than generic sorting methods.

Some smoothing image process algorithms use the median to output each pixel in a resultant smoothed image. For each output pixel, the median value is calculated considering its N neighbor pixels, where N is typically defined prior to the median calculation s. Being the images in gray scale, the contextual information in this example can indicate that the range of these pixels is between 0 and 255, while N is defined by the window size used to acquire the surrounding pixel neighbors. A specialization process in this method includes the selection of the best implementation for the given parameters, and an optimization phase that takes into account these values.

The code implementation of a median image processing filter presented in [Fisher et al. 2005] is an example of code developed to be generic (from a library). The code considers the use of an ArrayList to store the values to be sorted. Then, the algorithm determines the $\lceil \frac{N}{2} \rceil$ maximum values in the ArrayList and removes them. The last maximum value to be removed is the median value. This implementation may add

performance overhead in terms of the data structure as the remove operation of data elements of the ArrayList is heavier than when using a LinkedList and there might be more efficient algorithms to calculate the median and considering specific information regarding the execution environment.

We have analyzed different median algorithms and evaluated their performance when executing in a JRE 1.8.0_11 in a PC with Ubuntu 13.10, Intel® Core™ i5 @ 3,20GHz * 4, with 8 GB of RAM. Fig. 1 depicts the speedups achieved by code specialization according to the input parameters (window size) for the 6 most relevant approaches, namely, quicksort, counting sort, and a sorting network with different optimizations: unfolded (usn), optimized to determine the median value (usn_optimized), array accesses replaced with local variables (usn_localvars), and optimized version using local variables (usn_optimized_localvars). Our experiments show that when dealing with a small number of values (e.g., 9 values for a 3x3 window), a sorting network, optimized to output the median, is considerably faster than other approaches. On the other hand, for larger number of values the counting sort is faster, while the performance of the sorting network degrades significantly.

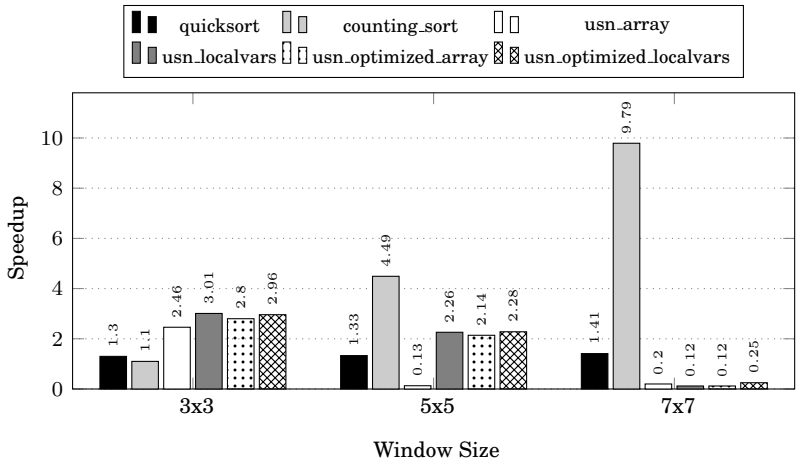


Fig. 1: Speedups achieved with code specialization for a smooth image processing algorithm using the median approach and for an image size of 1024x768 pixels.

In this example, we can see that runtime adaptability needs to consider the switch among different algorithms and possible specialized code versions for each algorithm. For instance, when considering the use of different sizes of unfolded sorting networks one possibility is to use a template-based generator of JVM instructions that is able to provide the specialized versions of the sorting network in runtime.

## 3. RUNTIME ADAPTABILITY USING LARA

A DSL, based on an AOP approach, may allow the developers to use a language focused on modifying a target application, without changing the source code, with goals that range from code monitoring to specialization [Cardoso et al. 2012]. The DSL can provide low-level constructions allowing rigorous definition of which modifications a strategy should apply and where it should take place. It also provides high-level constructs in which several low-level constructions may be invoked with user-defined heuristics.

Following an AOP approach allows developers to have more control on the application of secondary concerns (e.g., optimization of specific kernels with specific parameters), promoting, among other features, program maintainability, program portability and developer productivity. A DSL able to code adaptive behavior, allows developers to specify strategies for adaptation, improves portability, and helps tools to map those strategies to the target system.

The target application remains independent from this approach for its execution. Our approach is driven by three phases. First, the runtime adaptation strategies are defined in a DSL. The application is then weaved, at compile-time, with the given strategies, to comprise runtime adaptability. During the execution of the application, execution points trigger runtime weaving and apply the adaptation strategy.

For the specification of runtime strategies, we are extending the LARA language[Cardoso et al. 2012] to include runtime directives, in order to make runtime information accessible in the weaving process. The new LARA extension provides dynamic functionality to the language, and access to runtime information within an apply statement. As an example, a strategy for the selection of the best median implementation, according to Section 2, is depicted in Fig. 2. This strategy selects the required parameters, at runtime, and dynamically applies a specialization based on those parameters. Note that this strategy assumes the existence of a call to method ”getMedian” in method ”smooth” recieving an ”inputImage” as argument, and the size of the processing window is defined as a field in an ”ImageUtils” class.

```
1  aspectdef Best_median_implementation
2   select function{"smooth"}.call{"getMedian"} end
3   select dynamic
4    windowSize: class{"ImageUtils"}.field{"windowSize"};
5    image: function{"smooth"}.argument("inputImage");
6   end
7   apply dynamic
8    switch(&windowSize){
9     case 3: $call.perform specialize("sortedNetwork.tpl", &window.size); break;
10    // case 5 ...
11    case 7: if(&image.range == "[0,255]")
12        $call.perform specialize("countingSort.tpl", &window.size, &image.range);
13      else $call.perform specialize("quicksort.tpl", &window.size); break;
14   }
15  end
16 end
```

Fig. 2: LARA strategy example for runtime specialization.

Fig. 3 shows the current flow of our concept. At compile time the weaving engine receives the application to be adapted and the LARA strategies ①. The application is adjusted according to where adaptation is intended, and where the required data can be retrieved, i.e., execution site where the necessary runtime information is defined. The weaving engine generates an adapted version of the application. The adaptive engine is then supplied with template-based generators ② and the adapted application is executed in the JVM ③. The adapter is responsible to generate specialized code versions with the provided templates, according to the given input runtime information and strategy. During program execution, when an execution point triggers an adaptation event, it generates a new specialized version according to the given input from the application ④. The new version is then executed instead of the original version ⑤.

Considering the strategy in Fig. 2, the adaptation phase will use the given template and the information retrieved from the dynamic execution point to select the best im-

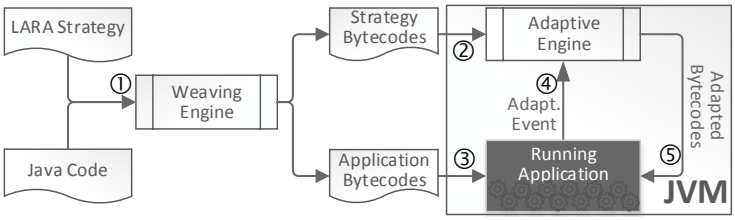


Fig. 3: Framework and flow for the runtime adaptation approach.

plementation and apply compiler optimizations using this information. For instance, for the median example, a sort with 3x3 elements and based on the previous results, the engine would select the sorting network as the best implementation and apply a set of optimizations, to produce the specialized version, namely: complete loop unrolling, scalar replacement, and optimize to retrieve only the median.

Fig. 4 presents part of the template-based sorting network code generation, before (a) and after (b) the specialization. For the sake of simplicity, we present Java code instead of JVM instructions (present in the Java bytecodes). The template-based generator fully unrolls the loops, the array accesses are replaced with accesses to local variables and the unnecessary instructions were removed. The template used here considers that the input values are stored in an array of integer values and is able to generate the unfolded code needed to calculate a median of $N$ (variable length in the code) values based on an odd-even sorting network.

```
for(int i = 0, init = 0; i < length/2; i++){
 for(int index=init; index < length-1; index+=2){
  if(values[index] > values[index+1]) {
   int temp = values[index];
   values[index] = values[index+1];
   values[index+1] = temp;
  }
 }
  init = 1-init; //for odd-even swaps
}
...
if(length % 2 == 0) { //length is even
 int middleLeft = values[values.lenght/2 - 1];
 int middleRight = values[values.lenght/2];
 return (middleLeft + middleRight)/2;
}
return values[values.length/2];
```

(a) Original, folded, sorting network

```
int value_0 = values[0];
int value_1 = values[1];
//6 lines omitted
int value_8 = values[8];
if (value_0 > value_1) {
 int temp = value_0;
 value_0 = value_1;
 value_1 = temp;
}
// +/-140 lines omitted
if (value_3 > value_4)
 value_3 = value_4;
if (value_5 > value_6)
 value_5 = value_6;
if (value_4 > value_5)
 value_4 = value_5;
return value_4;
```

(b) Specialized version

Fig. 4: Template-based specialization in the sorting network algorithm.

## 4. EXPERIMENTAL RESULTS

We present here some of the experiments we have conducted in order to evaluate the potential for runtime specialization. We include as benchmarks an image median filter, the calculation of the median, the sorting operation, and a Sobel image processing operation. All the experiments were performed in a JRE 1.8.0_11 JVM in a PC with Ubuntu 13.10, Intel® Core™ i5 @ 3,20GHz * 4, with 8 GB of RAM.

The experiments we include here are all based on the changing of a critical operation of an algorithm with a more suitable version for a specific runtime context (usually de-

pendent of specific data values). They make strong evidence of the performance gains we can achieve when commuting to those more suitable implementations. The following are the versions used in the experiments:

- **ifu:** full unroll of the innermost loops
- **local_filter:** move the filter coefficients used in an operation to the convolution method
- **ifu_sr:** *ifu* with replacement of array accesses to the filter coefficient with the corresponding scalar value
- **ifu_sr_dr:** *ifu_sr* and reuse of N neighbour elements (e.g., 6 elements reused for a window of 9)
- **usn:** unfolded sorting network

**Sobel Image Processing:** This experiment uses the Sobel algorithm [SOBEL 1990], which is an edge detection process for images. The version used in this example consists of three phases:

1) Gaussian convolution to smooth the input (gray) image
2) Convolution with a vertical Sobel kernel
3) Convolution with an horizontal Sobel kernel

All the three operations are using the same convolution method, where the only difference between them is the window coefficients. Because of this, we are able to specialize the convolution method for each of these operations, considering three different versions of this method.

We tested different transformations to verify how the program behaves. Initially, knowing the kernel size, we performed a full loop unroll on the innermost loops that iterate over the window coefficients. In the second transformation we moved the kernel inside the convolution method as a local variable, instead of as a parameter. Because of this, we developed a different method for each convolution operation. The third transformation builds on top of the first transformation. We used scalar replacement to substitute the accesses to the array of coefficients with the corresponding values. Furthermore, being some coefficients equal to zero, some statements such as $sum+ = image[x][y] * coefficient$ were eliminated. Finally, the fourth transformation starts from the previous and reuses data from a given iteration that is necessary in the following iteration. The results for each of these transformations are presented in Fig. 5 (a), where the last series represents the execution of the entire Sobel algorithm, with the three operations.

The convolution specialization for each type of operation provides good individual speedups. Simply fully unrolling the inner loops allows more than 60% speedup. The replacement of the array accesses to the kernel with the corresponding coefficients is an important optimization in this specialization, removing the accesses overhead. This optimization provides speedups from a factor of 4 to 6.5. As mentioned before, when the coefficient is zero, the operation is ignored by the generator, which results in less arithmetic and assignment operations executed. This advantage can be observed when comparing the *ifu_sr* speedups between the vertical and horizontal convolution with the gaussian convolution (white bars in Fig. 5 (a)).

Data reuse provides a slight boost from the previous optimization. Fig. 6 shows the reuse scheme applied in each iteration. For a 3x3 kernel, this transformation allows reusing 6 values from the previous iteration, meaning that only 3 array accesses are necessary (9 array accesses in the original version). The horizontal convolution takes advantage of not requiring the middle row of elements, hence only accessing and reusing the top and bottom rows (see Horizontal Conv. in Fig. 6). This means that we reuse 4 values and perform only 2 array accesses. In the vertical convolution, for a

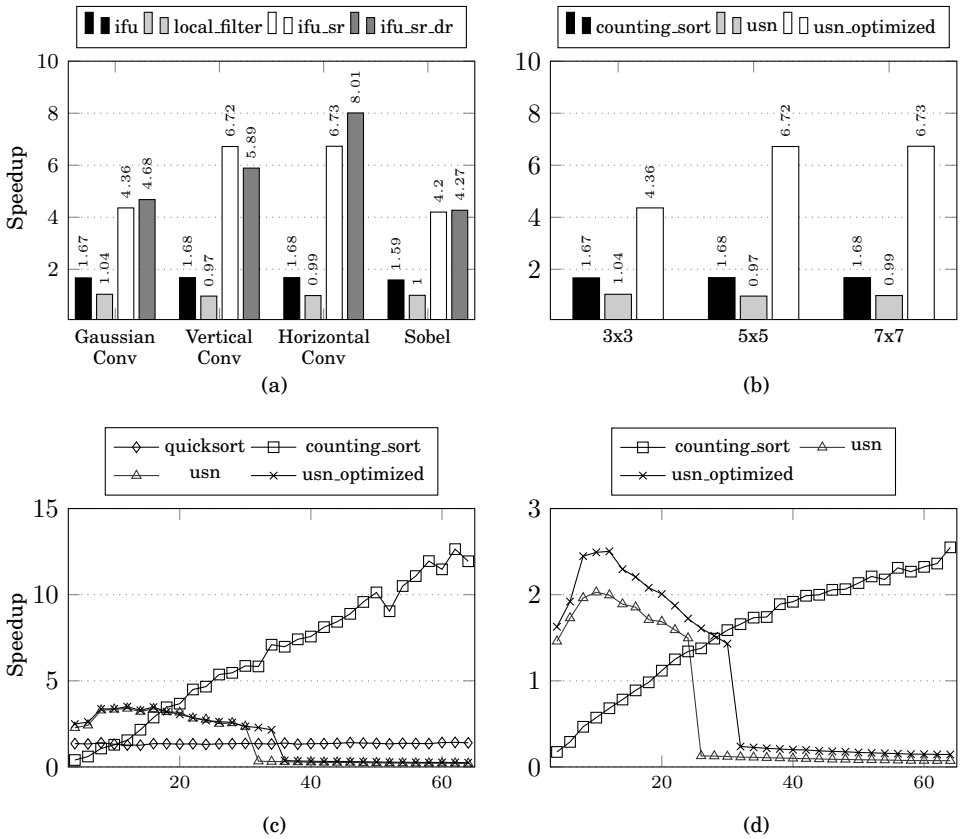


Fig. 5: Results for all the experiments. These represent the speedup: **(a)** when using specialized versions for the Sobel operations, **(b)** when inlining the versions presented in the motivational example (please see Fig. 1), **(c)** over the original sorting algorithm when finding the median of a given number of elements and **(d)** over the quicksort algorithm when sorting an array for a given number of elements.

given iteration, the middle column is not necessary (see Vertical Conv. in Fig. 6). However, since we are iterating from left to right, iteration $i + 1$ requires us to save the middle column (as it will be used in iteration $i + 2$). Therefore, the vertical convolution is not able to attain the same benefits as the horizontal convolution, requiring three array accesses and reusing 3 values.

The Sobel operation is the sequential execution of the first three operations. This means that Sobel was executed with the specialized versions of the other three operations. By applying the scalar replacement (with or without data reuse), we are able to achieve an overall speedup of 4.

**Inlined Image Median Smooth Filter:** Following the experiment presented in Fig. 1, Section 2, we now show the results of inlining algorithm versions to compute the median of a set of pixels in the median filter. This transformation avoids the need of storing the pixel neighbors in an array and send them to the method responsible to calculate the median of a set of pixels.

The unfolded sorting network (USN) presented in this example (usn and usn_optimized) are the same as the USN versions using local variables whose results

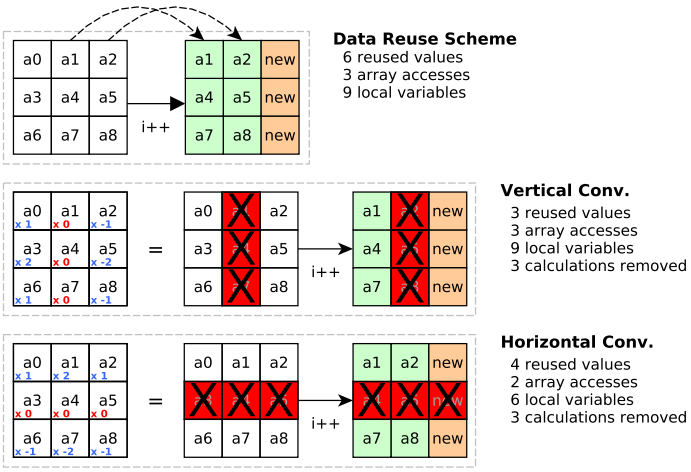


Fig. 6: Data reuse scheme used in the Sobel algorithm and its implications for the vertical and horizontal convolution operations after scalar replacement and useless code elimination are performed.

are presented in Fig. 1. The inlined version of the counting sort uses the neighbor values directly in the counting array, avoiding half of load/stores. The inlined USN copies the neighbor values directly to the local variables, saving $2/3$ of load/store operations. Fig. 5 (b) depicts the speedups attained with these inlined versions, over the inlined original version (based on the Java code in [Fisher et al. 2005]).

Comparing to the results presented in Fig. 1, an inlined version provides better performance in an USN for a 3x3 kernel, maintaining the same speedups for the other algorithms and kernel sizes. The speedups for counting sort maintain the same pattern compared to the outlined version. In the inlined versions (Fig. 5 (b)), there is no performance gain when using the optimized USN, as it seems that the JVM HotSpot is able to remove the unnecessary operations.

**Median Detection:** In this experiment we test alone the median operation used in the median filter for different array sizes, to observe if the performance improvements behavior changes with the number of elements. Fig. 5 (c) illustrates the speedups over the original version (i.e., the median used in the image processing library [Fisher et al. 2005]).

The quicksort algorithm maintains a steady speedup (between 25% and 40%) for every array size tested, as seen in Fig. 1. This algorithm may be useful for larger number of elements, and when we are unable to use counting sort, for instance, for larger values ranges of the elements to be sorted. A USN is the best approach for lower number of elements, while the counting sort is more effective for larger number of elements. Compared to the original version, counting sort is able to continuously improve the performance, as this algorithm has linear complexity while the original has $N \cdot \log N$ complexity.

**Sort Algorithms:** Here, we evaluate how the performance of the sorting algorithms used when varying the input dataset size. Fig. 5 (d) presents the speedups achieved by the counting sort and USN algorithms over the quicksort implementation. The algorithms behavior is similar to the previous tests, where the USN versions are the best solution for small number of elements, while the counting sort algorithm is the best solution as the number of elements grows. For the USN, by using local variables we

achieve better performance than when using arrays, maintaining greater performance for a slightly larger number of elements.

The USN versions show a sharper performance drop than in previous experiments since we are comparing them to the quicksort algorithm, which, in turn, maintains a steady speedup over the original version.

**Analysis:** The Sobel specialization depicts the advantage of program specialization based on contextual information. With an array of coefficients, we are able to generate specialized versions that outperform the original version, applying optimizations not performed by the JIT. The speedups obtained from the experiments with the median and sort algorithms show the advantage on algorithm selection taking into account the input datasets. These results provide evidence of the high impact on considering a specialization/optimization layer, as the JIT may not be able to perform more agressive optimizations (scalar replacement and data reuse in this case). Our approach aims for a higher-level layer than JIT in order to provide these type optimizations.

## 5. RELATED WORK

ADAPT [Voss and Eigemann 2001] is a compiler-support framework which provides dynamic and adaptive optimization processes, making use of compiler optimizations and accessible optimization tools. The compiler generates application-specific runtime systems based on the target application, optimizations required and heuristics to apply them. The framework supports dynamic compilation, allowing one to explore different implementations through “runtime sampling”.

The elastic computing [Wernsing and Stitt 2010] concept is based on using specialized functions that allow an optimization framework to try different implementations,possibly using different algorithms. They use a library of specialized functions, a tool for implementation planning and a runtime environment system to combine with a given application code. The user only specifies the function to be used, and the framework is responsible to dynamically choose the best implementation.

The rePLay framework [Patel and Lumetta 2001] aims to improve performance application using execution-guided optimizations, combining instruction pattern with branch prediction. This framework takes advantage of runtime stability, i.e., instruction and data patterns that repeat during a program execution. These patterns are used to generate optimized frames (single-entry, single-exit code regions) that replace the execution of normal, conditioned instruction sets.

The deGoal tool [Charles et al. 2014] embeds dynamic code generators into applications, providing runtime data-dependent optimizations for the target processing kernels. The applications kernels are built in fast, portable, and small binary code generators called “compilettes”. A compilette generates ad hoc versions of a kernel code at runtime that are optimized to the current program/system situation.

Khan et al. [Khan et al. 2008] propose an automated approach to deliver specialization at runtime, overcoming the code size and runtime activities overhead with a hybrid specialization approach, by first generating optimized code, through compile-time specialization, and then generate templates working with a set of input values, through runtime specialization, performed for a small number of instructions, in a binary template. During execution, the templates are adapted to the new values.

Our approach distinguishes from the previous ones with the efforts to define a DSL able to express runtime adaptivity strategies and the dynamic application of template-based code generators. We are focusing on the use of a DSL to express the application of runtime code optimizations, using, for instance, code specialization based on contextual information. Our approach, regarding runtime specialization/optimizations, is orthogonal to the work on JITs [Aycock 2003] as we provide a specialization layer, focused on adaptive strategies based on pre-optimized/specialized bytecodes. The adap-

tivity is envisioned as strategies expressed in LARA that at runtime decide among specialized/optimized versions.

## 6. CONCLUSION

This paper proposed an aspect-oriented programming approach for runtime adaptability considering the dynamic specialization of some parts of algorithms by using code versions embodied in templates. Those templates are then used to generate code at runtime. The approach relies on using a domain-specific language to program adaptation strategies decoupled from the target application. Adaptation strategies allow runtime specialization based on contextual information, by which is possible to apply compiler optimizations more efficiently or even decide which implementation is adequate for the current execution state. We believe that our approach will provide higher program maintainability and portability, and the exploration of different strategies in early development cycles. Our preliminary experiments reinforce the benefits of code specialization using contextual information. As ongoing and future work, we intend to study different low-overhead techniques for efficient runtime adaptation and to investigate more examples to which a template-based code generator approach is suitable.

## REFERENCES


AYCOCK, J. 2003. A brief history of just-in-time. *ACM Comput. Surv. 35,* 2, 97–113.

CARDOSO, J. M., CARVALHO, T., COUTINHO, J. G., LUK, W., NOBRE, R., DINIZ, P., AND PETROV, Z. 2012. Lara: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 179–190.

CHARLES, H.-P., COUROUSSÉ, D., LOMÜLLER, V., ENDO, F. A., AND GAUGUEY, R. 2014. degoal a tool to embed dynamic code generators into applications. In *Compiler Construction*. Springer, 107–112.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C., ET AL. 2001. *Introduction to algorithms*. Vol. 2. MIT press Cambridge.

DINIZ, P. C. AND RINARD, M. C. 1997. Dynamic feedback: an effective technique for adaptive computing. *SIGPLAN Not. 32,* 5, 71–84.

FISHER, R., PERKINS, S., WALKER, A., WOLFART, E., BROWN, N., CAMMAS, N., FITZGIBBON, A., HORNE, S., KORYLLOS, K., MURDOCH, A., ET AL. 2005. Hipr2: Image processing learning resources.

FLINN, J. AND SATYANARAYANAN, M. 2004. Managing battery lifetime with energy-aware adaptation. *ACM Trans. Comput. Syst. 22,* 2, 137–179.

KHAN, M. A., CHARLES, H.-P., AND BARTHOU, D. 2008. An effective automated approach to specialization of code. In *Languages and Compilers for Parallel Computing*. Springer, 308–322.

KISTLER, T. 1997. *Dynamic runtime optimization*. Springer.

MUKHIJA, A. AND GLINZ, M. 2005. *Runtime adaptation of applications through dynamic recomposition of components*. Springer-Verlag.

PATEL, S. J. AND LUMETTA, S. S. 2001. replay: A hardware framework for dynamic optimization. *Computers, IEEE Transactions on 50,* 6, 590–608.

SOBEL, I. 1990. An isotropic 33 image gradient operator. *Machine Vision for three-demensional Sciences*.

STITT, G., LYSECKY, R., AND VAHID, F. 2003. Dynamic hardware/software partitioning: a first approach. In *Proceedings of the 40th annual Design Automation Conference*. ACM, 250–255.

VAHID, F., STITT, G., AND LYSECKY, R. L. 2008. Warp processing: Dynamic translation of binaries to fpga circuits. *IEEE Computer 41,* 7, 40–46.

VOSS, M. J. AND EIGEMANN, R. 2001. High-level adaptive program optimization with adapt. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. PPoPP '01. ACM, New York, NY, USA, 93–102.

WERNSING, J. R. AND STITT, G. 2010. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *ACM SIGPLAN Notices*. Vol. 45. ACM, 115–124.