First Workshop on Resource Awareness and Application Autotuning in Adaptive and Heterogeneous Computing (RES4ANT)

Adaptive and heterogeneous computing platforms are gaining interest for applications spanning from embedded to high performance computing due to their promising power/performance ratio. However, sharing hardware resources creates some challenges with respect to predictable execution time and power consumption. In traditional real-time approaches, resource usage is over dimensioned to achieve worst case guarantees, whereas in best effort approaches, predictability remains a challenge. The goal of the workshop is to bring together researchers from the area of resource awareness and application autotuning, to discuss their various approaches, their commonalities and differences, to foster collaboration between them and to share their most recent research achievements with the international research community.

ORGANISATION

General Co-Chairs:	Cristina Silvano, Politecnico di Milano, Italy
	Walter Stechele, TU Munich, Germany
	Stephan Wong, TU Delft, The Netherlands
Poster Session Chair:	Jeronimo Castrillon, TU Dresden, Germany
Panel Session Chair:	Michael Hübner, Ruhr-Universität Bochum, Germany
Web Chair:	Amir H. Ashouri, Politecnico di Milano, Italy

FORMAT OF THE EVENT

For the main workshop sessions we plan invited talks (30 minutes long) from top-level specialists coming from both industry and academia. We plan an open Call for Posters in order to give young researchers and PhD students a chance to introduce their research and to get in personal contact during the workshop. In addition to the PhD Forum poster session, there will be a short interactive presentation time for introducing the posters during the workshop. At the end of the workshop day, speakers from the main workshop sessions are invited to the one-hour panel on: "Resource Awareness and Application Autotuning: Challenges and Trends" to further discuss similarities and differences between the approaches, as well as the expected benefits and limitations of resource-aware computing and application autotuning.

TARGET AUDIENCE

DATE represents the major international event for research on Systems-on-Chip, Systems-on-Board and Embedded Systems Software. This workshop targets the traditional DATE community as well as experts from high performance computing, both working on resources awareness and application autotuning. Based on the experience of the organizers, we plan to enforce our publicity efforts to get a significant attendance (estimated as about 50 registered attendees from both industry and academia). To further encourage the attendance to the Workshop, we plan to distribute an open Call for Posters covering the topics of the workshop topic areas. Poster submissions should either be a 150-200 word abstract or in the form of the poster itself. Posters will be published online at the Workshop web site.

Timeline: Call for Posters online at the WS web site from Nov.1st. Poster submission deadline: Feb. 1st. Poster notification of acceptance: Feb. 20th. The advanced program for the Workshop will be finalized to be included in the DATE program of Workshops (before Dec. 1st), while posters program will be posted online by March 1st.

PRELIMINARY PROGRAM

8.45	**9.00**	**Opening Session**
9.00	**10.30**	***Morning Session 1***
9.00	**9.30**	Cathal McCabe, Xilinx: "Programing and benchmarking FPGAS with software-centric design entries"
9:30	**10.00**	Jürgen Teich, FAU Erlangen: "Adaptive Restriction and Isolation for Predictable MPSoC Stream Processing"
10.00	**10.30**	Introduction to the Poster Session
10.30	***11.00***	***Coffee Break***
11:00	**12.00**	***Morning Session 2***
11:00	**11.30**	Alexander Moskovsky, RSC Group: "Energy efficiency in high performance computing. Examples from RSC" experience
11.30	**12:00**	Axel Auwetter, Leibniz Supercomputing Centre: EU Projects MontBlanc and DEEP-ER
12.00	***13.00***	***Lunch***
13.00	**14.30**	***Afternoon Session 1***
13.00	**13.30**	João Cardoso, University of Porto: "A DSL-based Approach for Cross Layer Programming: Monitoring, Adaptivity and Tuning"
13.30	**14.00**	Axel Jantsch, TU Vienna: "Resource management in self-aware platforms"
14.00	**14.30**	**Poster Session** Interactive Presentations
14.30	***15.00***	***Coffee Break***
15.00	**17.00**	***Afternoon Session 2***
15.00	**15.30**	Andreas Rohatschek, Robert Bosch GmbH: "DRIVERS AND SOLUTIONS FOR TAILORED AUTOMOTIVE ECU ARCHITECTURES"
16.00	**17.00**	**Panel discussion on** *"Resource Awareness and Application Autotuning: Challenges and Trends".* Speakers from morning & afternoon sessions are invited as panelists.
17.00		Closing

FURTHER INQUIRIES

Cristina Silvano
Politecnico di Milano
silvano@elet.polimi.it
home.deib.polimi.it/silvano/

Walter Stechele
Technical University of Munich
Walter.Stechele@tum.de
www.lis.ei.tum.de/?id=stechele

Stephan Wong
TU Delft
j.s.s.m.wong@tudelft.nl
www.ce.ewi.tudelft.nl/wong/

Resource-Aware Application Execution Exploiting the BarbequeRTRM

Giuseppe Massari, Simone Libutti,
William Fornaciari, Federico Reghenzani
and Gianmario Pozzi

Politecnico di Milano
DEIB: Dipartimento di Elettronica, Informazione e Bioingegneria
giuseppe.massari@polimi.it

Abstract. Energy efficiency and thermal management have become major concerns in both embedded and HPC systems. The progress of silicon technology and the subsequent growth of the dark silicon phenomena are negatively affecting the reliability of computing systems. As a result, in the next future we expect run-time variability to increase in terms of both performance and computing resources availability. To address these issues, systems and applications must be able to adapt to such scenarios. This work provides a brief overview of the Barbeque Run-Time Resource Manager (*BarbequeRTRM*) and the application execution model that it exploits, in order to deal with run-time performance and available resources variability.

1 Introduction

The need of *resource-aware* and *adaptive* applications is driven by several issues and requirements that are typical of modern computing systems. For instance, embedded mobile devices must deal with the limited energy budget provided by the battery, while HPC centers must afford huge costs due to the power consumption and the cooling of the infrastructure. Furthermore, the *dark silicon* phenomenon affecting modern processors is becoming prominent[1], since it is increasing the amount of silicon area that must be turned off, to guarantee the power envelope of the processor. For all these reasons, a continuous and full usage of the whole set of system computing resources is often impossible to achieve.

On the application side, we can gain efficiency by implementing suitable adaptive behaviors like enabling/disabling the execution of a task, or scaling the accuracy of the output depending on the availability of computing resources. A run-time resource management framework can implement such approach by constraining the resource allocation according to system level requirements or runtime conditions, and providing to the applications suitable interfaces to check and negotiate the resource assignment.

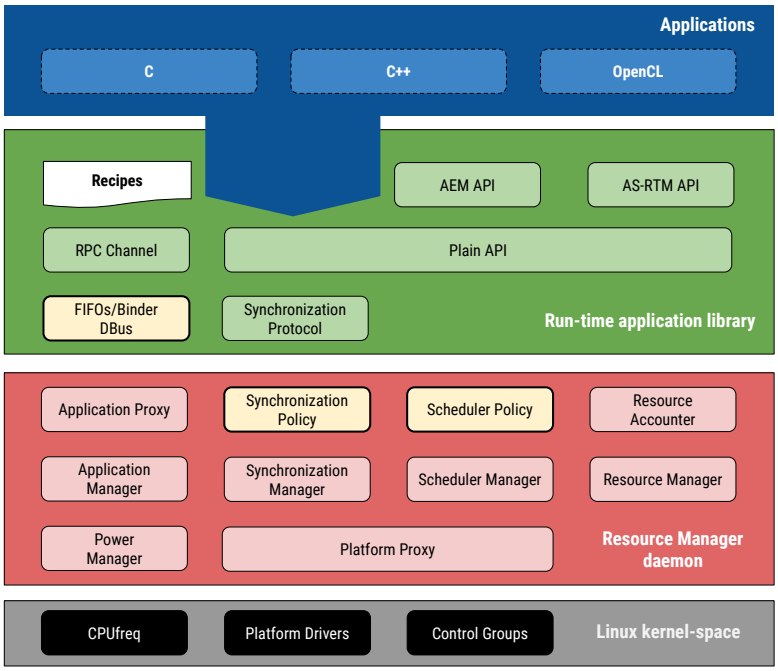


Fig. 1. The BarbequeRTRM Architecture. On top the programming languages supported by the application Run-Timr Library (RTLib). In red the resource manager core, on top of the support provided by the Linux OS to control the system resources.

2 Run-time Resource Management

The *BarbequeRTRM* is a modular and portable run-time resource manager targeting both embedded and High-Performance Computing (HPC) systems. From the hardware resources perspective, the framework can manage homogeneous and heterogeneous multi-core processors, as well as heterogeneous systems including devices characterized by completely different ISA (e.g., CPU and GPU).

The *modularity* of the BarbequeRTRM comes from a software architecture in which we can distinguish between *core* components and *plugin* modules. Typically, the latter are platform-specific extensions and selectable resource management policies.

The *portability* instead, is guaranteed by the exploitation of some underlying Linux operating system frameworks, like `cpufreq` and `cgroups`, that allows the *BarbequeRTRM* to enforce the resource allocation decisions [2].

2.1 Abstract Execution Model

The resource manager exposes its services to the applications through a run-time library (*RTLib*). The library accomplishes a two-fold objective: 1) to provide a

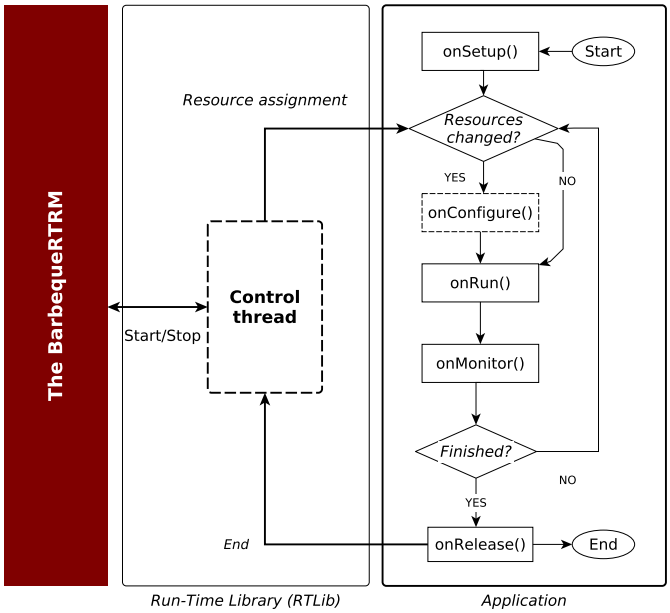


Fig. 2. Abstract Execution Model

communication channel between the resource manager and the applications; 2) to expose an execution model to support the implementation of the resource-aware adaptive execution of the applications[3].

In Figure 2 we show the Abstract Execution Model (AEM), that the run-time manageable applications must implemented accordingly. This execution model is put in place by defining and implementing a suitable C++ class, derived from the `BbqueEXC` class provided by the RTLib.

At run-time, the `BbqueEXC` member functions are called by a control thread, which is responsible of synchronizing the application execution with the decisional process of the resource manager. The rationale behind each member function implementation is the following:

`onSetup()`: setting up the application (initialize variables and structures, starting threads, ...). `onConfigure()`: check the amount of assigned resources and configure the application accordingly. `onRun()`: single cycle of computation (e.g., computing a single frame during a video encoding). `onMonitor()`: performance and QoS monitoring. `onRelease()`: cleanup and termination code.

Therefore, once the application ends the initialization step (`onSetup`), the control thread waits for the resource allocation decision coming from the BarbequeRTRM. As soon as it has been received, the `onConfigure` function is called. In this function, the application can then check the amount of assigned resources, and configure itself accordingly, before starting (or continuing) the execution, as sketched here below.

```
RTLIB_ExitCode_t BlackscholesEXC::onConfigure(int8_t awm_id){
    // Get the number of CPU cores assigned
    GetAssignedResources(RTLIB_ResourceType::PROC_NR, nr_cpu);

    // Configure ...
}
```

The functions `onRun` and `onMonitor` are then sequentially called and executed in a loop, until the entire computation is over.

The RTLib estimates the current performance of the application, in terms of *cycles-per-second (CPS)*, such that the application could check the gap between the required performance level and the one currently achieved. After that, the application can notify the resource manager about this gap.

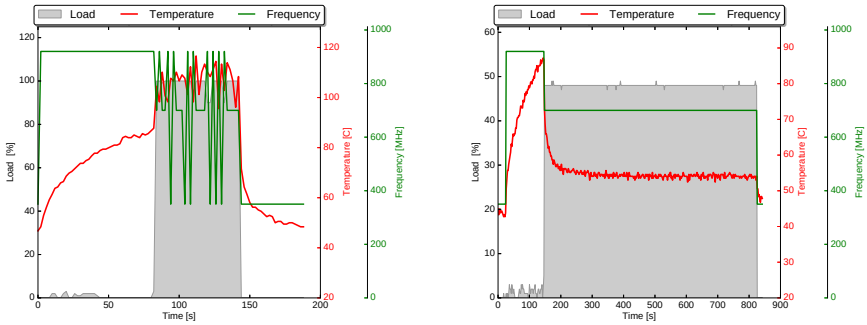
Considering also that the performance goal can vary depending on input data and external events, a effective approach is to exploit the `SetCPSGoal` function to specify the performance goal and the notification rate, as shown in the following example of `onMonitor` implementation:

```
RTLIB_ExitCode_t BlackscholesEXC::onMonitor() {
    // Specific event condition triggering the
    // change of performance requirements
    if (...)
      SetCPSGoal(2.5, 10);
    // ...
}
```

In the example, the application sets a performance goal of 2.5 CPS, and a notification rate of 10 cycles. The library keeps track of the application performance, computing the average CPS value over a (configurable) number of last execution cycles. Whenever the performance gap overcomes a given (configurable) threshold, such a gap value is sent to the resource manager. As a consequence, the amount of assigned resources can be adjusted accordingly. The notification rate is then exploited to bound the application reconfiguration rate, and hence the related overhead. In other words, the application asks the resource manager to send back a reconfiguration request after not less than 10 execution cycles or more.

3 Experimental Scenario

In this section we show results of the resource-aware adaptive execution of *blackscholes* from the PARSEC benchmark suite [4] on a embedded development board that features an ARM Cortex A9 dual-core CPU. The benchmark has been properly modified to fit the Abstract Execution Model. The frequency of the CPU has been set to its maximum value, which is 920 MHz. The full CPU usage, which is shown in Figure 3a, causes the chip temperature to raise over 100°C, thus triggering the thermal throttling response of the operating system.



(a) CPS = 4 (b) CPS = 1

Fig. 3. PARSEC *blackscholes* execution: CPU load, temperature and clock frequency variations according to two performance requirements: a) 4 cycles-per-second; b) 1 cycle-per-second.

A continuous frequency scaling is operated in order to cool down the CPU, with performance variability as a further consequence.

In Figure 3b, the application sets a performance goal of CPS=1. The resource manager takes into account such information shrinking the amount of CPU time assigned. The implicit result is a lower but more stable performance level, along with a reduced thermal stress.

References

1. H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108
2. P. Bellasi, G. Massari, and W. Fornaciari, "Effective Runtime Resource Management Using Linux Control Groups with the BarbequeRTRM Framework," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 2, p. 39, 2015.
3. G. Massari, E. Paone, P. Bellasi, G. Palermo, V. Zaccaria, W. Fornaciari, and C. Silvano, "Combining application adaptivity and system-wide resource management on multi-core platforms," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*. IEEE, 2014, pp. 26–33.
4. C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

A Scalable Black-Box Optimization System for Auto-Tuning VLSI Synthesis Programs

Matthew M. Ziegler[1], Hung-Yi Liu[2*], George Gristede[1], Bruce Owens[3], Ricardo Nigaglioni[4], and Luca P. Carloni[2]

1 IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
{zieglerm, gristede}@us.ibm.com
2 Department of Computer Science, Columbia University, NY, USA
{hungyi,luca}@cs.columbia.edu}
3 IBM Systems & Technology Group, Rochester, MN, USA
browens@us.ibm.com
4 IBM Systems & Technology Group, Austin, TX, USA
nricardo@us.ibm.com

Abstract. Modern logic and physical synthesis tools provide numerous options and parameters that can drastically impact design quality; however the large number of options leads to a complex design space difficult for human circuit designers to navigate. We tackle this parameter tuning problem with a novel system employing intelligent search strategies and parallel computing, thus automating one of the key design tasks conventionally performed by a human designer. We provide an overview of this system, called SynTunSys, as well as results from employing it during the design of the IBM z13 22nm high-performance server chip, currently in production. During this major processor design, SynTunSys provided significant savings in human design effort and achieved a quality of results beyond what human designers alone could achieve, yielding on average a 36% improvement in total negative slack and a 7% power reduction.

Keywords: synthesis · design space exploration · parameter tuning · optimization · VLSI design · design methodology

1 Introduction

The design of modern high-performance processors is a quest to optimally tune and balance multiple objectives, such as performance, power, and reliability. This multi-objective design space is further complicated by the need for more complex VLSI (very-large-scale integration) chips to fuel the ever increasing desire for more compute power. To cope with this design complexity, the VLSI design community has leveraged CAD (computer-aided design) tools for many decades now; however, the high flexibility and sophistication of advanced synthesis tools increases their complexity and makes navigating the design space difficult and sometimes non-intuitive for their users.

[*] Hung-Yi Liu is now with the Intel Design Technology & Solutions Group, Hillsboro, OR, USA.

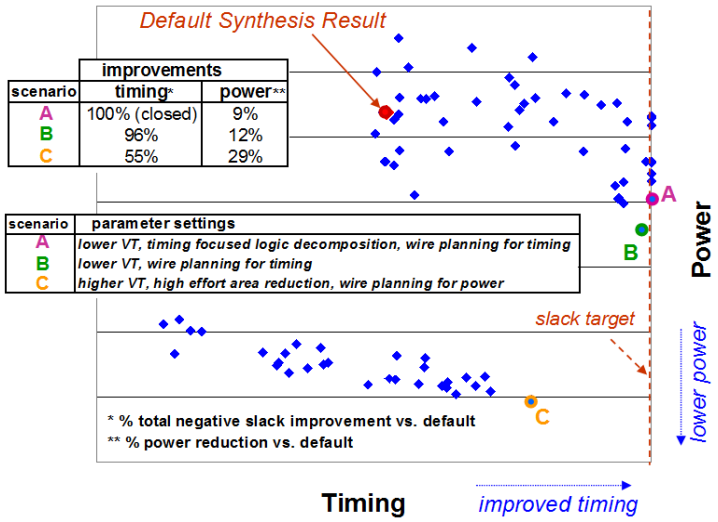


Fig. 1. An example of the available design space by modifying synthesis parameters.

The industrial synthesis tool-flow we employ has over 1000 parameters [1]. These parameters span the logic and physical synthesis space and the control settings for modifying the synthesis steps, such as: logic decomposition, technology mapping, placement, estimated wire optimization, power recovery, area recovery, and/or higher effort timing improvement. The parameters also vary in data type (Boolean, integer, floating point, and string). Considering that an exhaustive search of only 20 Boolean-type parameters leads to over one million combinations, and synthesis runs may take several hours or even days, it is clear that intelligent search strategies are required.

As an example of the wide design space available from modifying synthesis parameters, Fig. 1 shows the scatter plot of achievable design points for a portion of a synthesized floating-point multiplier macro. A macro may span from 1K to 1M gates in our context. Each point denotes the timing and power values achieved simply by tuning the input parameters of the synthesis program. The ultimate goal of this process is to reach timing closure at the lowest achievable power. Quite often the default values for the parameters are not ideal for a specific macro, which would benefit instead from parameter customization. Fig. 1 also highlights three scenarios (A, B, and C) along the Pareto frontier. These scenarios show the available tradeoffs between timing closure and power reduction, e.g., point A closes timing with a 9% power reduction, whereas point C improves timing by 55% with a 29% power reduction. These points along the Pareto set provide a number of potential steps towards the ultimate goal, depending on the additional techniques at the designer's disposal beyond parameter tuning. This example of a relatively simple macro underscores how significantly the parameters settings can affect a design.

2 Related Work

The synthesis parameter tuning problem we address can be classified as a black-box optimization problem, i.e., we treat the synthesis program as black-box software by supplying input conditions (input data and parameter settings) and measuring the

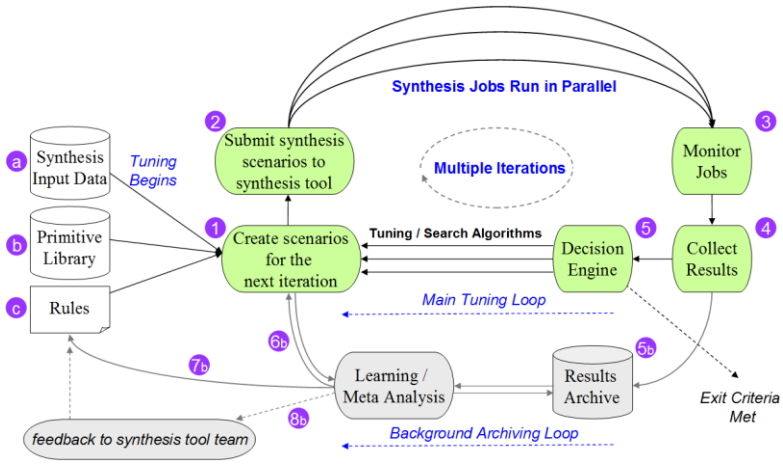


Fig. 2. Architecture of the SynTunSys process, which employs a parallel and iterative tuning process to optimize macros.

output response in terms of synthesis quality of results (QoR). Black-box problems are often approached using techniques from the field of simulation optimization [2], which is an umbrella term for optimization techniques that operate in the absence in of an algebraic model of the system. Considering each macro exhibits a unique input-output response to synthesis parameter settings and digital logic can take on an intractable number of functionalities, the synthesis tool-flow of our focus is far too complex to be modelled algebraically. Furthermore, our synthesis program often exhibits non-deterministic behavior, which is also a characteristic of many simulation optimization problems.

Black-box optimization techniques can also be employed for design-space exploration (DSE) purposes, however, unlike convention DSE, the goal of black-box optimization is often to find one or more optimal or near-optimal design points without necessarily requiring a complete exploration of the design space to determine the whole Pareto frontier of design tradeoff points.

Black-box optimization is a common problem seen across a number of fields, e.g., compiler tuning [3] and software engineering [4]. With respect to VLSI design, DSE is becoming a more attractive solution for complex problems across various levels of abstraction. At the architectural level, many DSE studies based on models or simulators have been used to explore multi-objective design spaces, e.g., [5]. Architectural-level studies, however, typically do not result in implemented designs. DSE approaches have also been applied in combination with high-level synthesis by a number of researchers, e.g., [6,7]. FPGA synthesis parameter tuning has been reported in [8] using genetic algorithms and in [9] using Bayesian optimization. However, prior to our work [10], we know of no publications targeting automated parameter tuning for logic and physical synthesis.

3 System Architecture

The framework of our parameter tuning system is shown in Fig. 2. SynTunSys consists of a main tuning loop that constructs synthesis scenarios consisting of synthesis parameter settings (Step (1)), submits and monitors synthesis jobs (2-3), analyzes the

Table 1. Average parameter tuning improvements over best known prior solution for a 22nm processor in production.

Pre / Post SynTunSys Comparison	Latch-to-Latch Slack	Total Negative Slack	Total Power
Improvement %	60%	36%	7%
Sum of 200 macros	(ps)	(ps)	(arb. units)
pre-tuning	-1929	-2150385	17770
post-tuning	-765	-1370731	16508

results (4), and iteratively refines the solutions (5). A second background loop archives the results of all runs from all macros, users, and projects, which can be mined for historical trends across multiple macros and to provide feedback in terms of the performance of synthesis parameters.

The SysTunSys cost function conveys the optimization goals. It converts multiple design metrics into a single cost number, allowing cost ranking of scenarios. Examples of available metrics include: multiple timing metrics, power consumption, congestion metrics, area utilization, electrical violations, runtime, etc. The selected metrics are assigned weights to signify their relative importance. The overall cost function is then a "normalized weighted sum" of the *m* selected metrics, expressed by Equation (1) where *Norm(Mi)* is the normalized *Mi* across all the scenario results in a SynTunSys run.

$$Cost = \sum_{i=1}^{m} W_i \cdot Norm(M_i) \qquad \begin{array}{l} where: \\ W_i = weight_i \\ M_i = metric_i \end{array} \qquad (1)$$

The SynTunSys decision engine algorithms are key components of the system that determine which scenarios should be run at each iteration, i.e., the decision engine tackles the parameter tuning black-box optimization problem discussed previously. The decision engine can also be upgraded independently and we constantly look to improve these algorithms in terms of QoR prediction accuracy and compute efficiency. Similar black-box tuning problems have been approached using a number of techniques, e.g., machine learning [3], Markov decision processes [11] and Bayesian optimization [9,12]. During the development of SynTunSys we have explored algorithms ranging from pseudo-genetic algorithms to on-line adaptive learning [13].

4 SynTunSys Results

SynTunSys was used during the design of the IBM z13 22nm server processor [14]. The processor underwent two chip releases (tapeouts) over a multi-year design cycle, during which SynTunSys was applied to macros over both releases. The chip consists of a few hundred macros that average around 30K gates in size, with larger macros in the 300K gate range. Prior IBM server processors have also used systematic parameter tuning on a smaller scale. The IBM POWER7+ [15] and POWER8 [16] processors employed an earlier version of SynTunSys during the second chip releases, but mainly for power reduction purposes. In the case of the z13 processor, however, Syn-

TunSys was employed during the entire design cycle for timing closure, power reduction, and improving macro routability.

Based on the efforts of a dedicated tuning team we were able to track SynTunSys results on approximately 200 macros from the processor core. Table 1 shows the average improvements achieved by SynTunSys over the best solution previously achieved by the macro owners.

Note that these results are based on the routed macro timing and power analysis; in most cases the best known prior solutions included manual parameter tuning by the macro owner. SynTunSys resulted in a 36% improvement in total negative slack, a 60% improvement in worst latch-to-latch slack (macro internal slack), and a 7% power reduction. The actual values of the metrics, summed across all the macros, underscores that the changes in the absolute numbers were significant, e.g., ~780,000 picoseconds of total negative slack was saved across ~200 macros.

5 References

[1] L. Trevillyan, et al., “An Integrated Environment for Technology Closure of Deep-Submicron IC Designs,” IEEE Design & Test of Computers, vol. 21:1, pp. 14-22, 2004.

[2] S. Amaran, et al., “Simulation Optimization: A Review of Algorithms and Applications,” 4OR - A Quarterly Journal of Operations Research, Dec. 2014.

[3] G. Fursin, et al., "Milepost GCC: Machine Learning Enabled Self-tuning Compiler," International Journal Parallel Programming, 39:296-327, 2011.

[4] A. Arcuri, G. Fraser, "Parameter Tuning or Default Values? An Empirical Investigation in Search-Based Software Engineering," Empirical Software Engineering, June 2013, Volume 18, Issue 3.

[5] O. Azizi, et al., "An Integrated Framework for Joint Design Space Exploration of Microarchitecture and Circuits," DATE 2010.

[6] S. Xydis, et al., "A Meta-Model Assisted Coprocessor Synthesis Framework for Compiler/Architecture Parameters Customization," DATE 2013.

[7] H.-Y. Liu and L. P. Carloni, “On Learning-Based Methods for Design-Space Exploration with High-Level Synthesis,” DAC 2013.

[8] M. K. Papamichael, P. Milder, J. C. Hoe, "Nautilus: Fast Automated IP Design Space Search Using Guided Genetic Algorithms," DAC 2015.

[9] N. Kapre, et al., “Driving Timing Convergence of FPGA Designs through Machine Learning and Cloud Computing,” FCCM 2015.

[10] M. M. Ziegler, H.-Y. Liu, G. Gristede, B. Owens, R. Nigaglioni, L. P. Carloni, “A Synthesis-Parameter Tuning System for Autonomous Design-Space Exploration,” DATE 2016.

[11] G. Beltrame et al., “Decision-Theoretic Design Space Exploration of Multiprocessor Platforms,” IEEE TCAD, 29(7):1083–1095, July 2010.

[12] Z. Wang et al., “Bayesian Optimization in High Dimensions via Random Embeddings,” Int’l Joint Conf. on Artificial Intelligence (IJCAI-13), 2013.

[13] M. M. Ziegler, H.-Y. Liu, L. P. Carloni, “Scalable Auto-Tuning of Synthesis Parameters for Optimizing High-Performance Processors,” ISLPED 2016.

[14] J. D. Warnock, et al., “22nm Next-Generation IBM System z Microprocessor,” ISSCC 2015.

[15] M. M. Ziegler, G. D. Gristede, V. V. Zyuban, “Power Reduction by Aggressive Synthesis Design Space Exploration,” ISLPED 2013.

[16] M. M. Ziegler, et al., “POWER8 Design Methodology Innovations for Improving Productivity and Reducing Power,” CICC 2014.

Using Reference Attribute Grammar-Controlled Rewriting for Runtime Resource Management

Johannes Mey[1], René Schöne[1], Daniel Langner[1], and Christoff Bürger[2]

[1] Chair of Software Technology, TU Dresden, Germany
johannes.mey@tu-dresden.de, rene.schoene@tu-dresden.de,
daniel.langner@mailbox.tu-dresden.de
[2] Department of Computer Science, Faculty of Engineering, Lund University, Sweden
christoff.burger@cs.lth.se

Abstract. To make distributed systems resource aware and adaptive, they can be modeled as self-adaptive systems. Such systems have a view of their own state and context, which can be represented by a model that is continuously updated and analyzed at runtime. However, such analyses need to be concise and efficient to allow large models and high adaptation rates. To achieve this, we apply *reference attribute grammar controlled rewriting* to implement the runtime model of a distributed task-scheduling case study for energy optimization.

1 Modeling Self-Adaptive Systems

Self-adaptive systems [1] are used to cope with changing requirements and contextual information at runtime. Furthermore, they need to provide short response times while maintaining low resource consumption and a convenient way to specify their internal state and algorithms. Another challenge is the high update rate of their context information[2]. Self-adaptive systems usually employ a feedback loop, e.g. *MAPE-K* [3], and have representation of their context, e.g. a runtime model. The *models@run.time* approach [4] uses models not only during development but also as a data representation at runtime. It has been shown that auto tuning and resource awareness can save energy in Big Data scenarios [5]. Our use case is a small, yet scalable, distributed Big Data scenario on a network of embedded devices. We use a self-adaptive system built around a runtime model, which is easy to specify, and can run algorithms efficiently with regard to response time. It employs grammar-based modeling and analysis to deal with frequent model updates efficiently.

2 Attribute Grammars for Runtime Models

Our solution uses *reference attribute grammars* [6] (*RAGs*) as its underlying technology. RAGs originate from the area of compiler construction to describe abstract syntax trees of program code. However, their intrinsic advantage – incremental evaluation – fits well to the described problems. Using RAGs, we

describe the structure of runtime models with a context free grammar that is well-suited for hierarchical structures. Non-hierarchical parts of a model can be described as well using *reference attributes* forming arbitrary overlay graphs. The analysis of runtime data is done using *attributes*, which are defined declaratively for specific non-terminals, achieving a concise specification.

However, the aforementioned updates of the runtime model hinder incremental evaluation commonly used in RAG systems since they rewrite the AST and therefore invalidate all previously performed analyses. This work uses a novel approach called *RAG-controlled rewriting* (*RACR*) [7], which treats model changes as term rewrites [8]. This enables the tracking of dynamic dependencies between attributes and the model, and thus incremental evaluation across model changes. Therefore, constantly changing self-adaptive systems can be analyzed efficiently, thus allowing a more frequent analysis and larger model sizes.

3 Runtime Models with RACR

RACR works in a three-phase process. In the first phase, a context free grammar with inheritance describing the runtime model is specified, like the one depicted below for our case study presented in section 4. Terminals are in lowercase and non-terminals in title case optionally suffixed by an alternative name and a colon.

```
Root ::= scheduler switching CompositeWorker
AbstractWorker ::= id state timestamp
CompositeWorker:AbstractWorker ::= AbstractWorker*
Switch:CompositeWorker ::=
Worker:AbstractWorker ::= devicetype Queue:Request*
Request ::= id size deadline dispatchtime
```

The second phase involves the attribution, that is the specification of attributes for certain non-terminals. Below, the `schedule` attribute defined for `Root` is listed. It reads the terminal `scheduler` and invokes an attribute to find an insertion position. All attributes and rewrites are written Scheme, using the API functions `ast-child`, `create-ast` and `att-value` to get a certain child of a AST node, create a new AST node and call an attribute, respectively.

```
(ag-rule schedule
  (Root (lambda (n time work-id load-size deadline)
    (att-value (ast-child 'scheduler n) n
               time work-id load-size deadline))))
```

At runtime, the system is performing rewrites and attribute evaluations in turns. Rewrites, like the one shown below, change the model and invalidate cached attribute values. If those attributes are called, RACR ensures their re-evaluation.

```
(rewrite-insert
  (ast-child 'Queue worker) ;list node to insert into
  index                     ;position of insertion
  (create-ast 'Request (list id size deadline #f)))
                            ;subtree for the new request
```

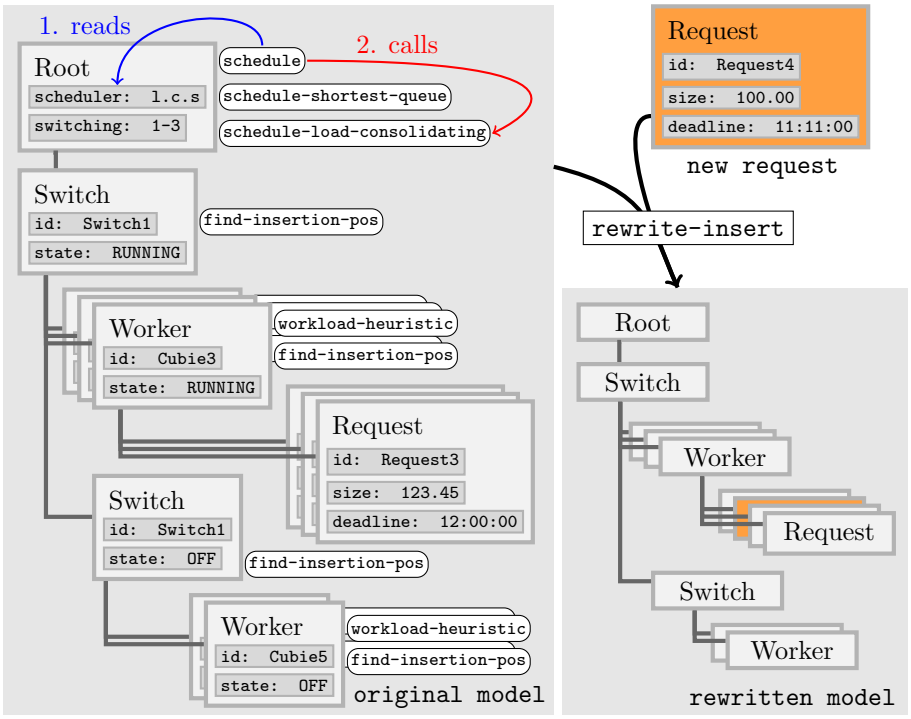


Fig. 1: Scheduler selection and scheduling of a request. Terminals are contained in non-terminal boxes, some selected attributes are attached. Terminals not relevant for the example are left out, `l.c.s` is the load consolidating scheduler.

4 A Case Study

To investigate the applicability of RACR to self-adaptive systems, we implemented the distributed indexing of Wikipedia pages using a network of system-on-a-chip workers [9]. These are Cubieboards having a 1 Ghz CPU, 1GB of RAM, running Linux, and are connected to a master via switches and Ethernet links. Every worker and switch is powered by a USB charging hub, which enables the switching and energy measurement of individual devices.

We developed an adaptation and two scheduling strategies, each written with RACR. The adaptation strategy controls the number of powered on workers. Our solution checks periodically for idle workers to be switched off while ensuring a minimum number of online workers to secure stable performance in case of load peaks. A round-robin scheduler always chooses the shortest queue, and a load-consolidating scheduler tries to use as few workers as possible.

The solution is evaluated in a small-scale case study, whose structure and the scheduling of a request on it are depicted in Figure 1. To analyze our approach's scalability, we developed a simulation environment that simulates the execution of

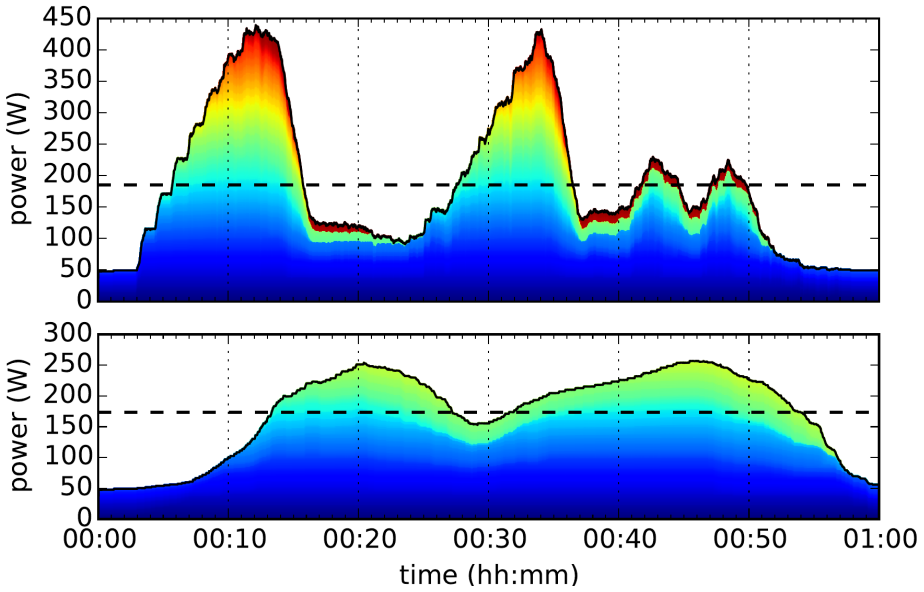


Fig. 2: Power consumption when scheduling a workload with a round-robin (top) and a load-consolidating (bottom) scheduler. The dashed line shows the average power consumption.

tasks and their associated energy consumption based on models we acquired from our physical use case. The simulated use case comprises 315 workers connected via 63 switches fulfilling 4,600 requests within an hour. Figure 2 shows the power consumption of the two scheduling strategies, using a different color for each worker. The load-consolidating scheduler (shown at the bottom) uses 6.4% less energy than the round-robin scheduler while using less workers.

As the model can be modified with rewrites, it permits addition and removal of workers and the exchange of scheduling and adaptation strategies during runtime.

5 Conclusion and Outlook

In this work, we showed the applicability of *RAG-controlled rewriting* for self-adaptive systems in a distributed data processing use case. In addition, we plan to conduct more case studies exploring the scalability and adding heterogeneity. Another case study involves an extended runtime model with included software structure, in which the presented concepts are applied to iteratively transform the model to code describing a constraint problem. First measurements show very short response times for every transformation after the initial one.

In conclusion, *RACR* enables incremental evaluation for large runtime models with high update rates, hence offering opportunities for the usage in adaptive, resource aware systems, such as Big Data systems.

Acknowledgments

This work is partly supported by the German Research Foundation (DFG) in the SFB 912 "Highly Adaptive Energy-Efficient Computing", the cluster of excellence cfaed, and within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907).

References

1. R. Lemos *et al.*, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS, R. Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. Springer, 2013, vol. 7475.
2. Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar, "Implicit Self-adjusting Computation for Purely Functional Programs," in *ICFP*. New York, NY, USA: ACM, 2011.
3. J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, Jan. 2003.
4. G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, Oct. 2009.
5. S. Götz, T. Ilsche, J. Cardoso, J. Spillner, T. Kissinger, U. Aßmann, W. Lehner, W. E. Nagel, and A. Schill, "Energy-Efficient Databases Using Sweet Spot Frequencies," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2014.
6. G. Hedin, "Reference attributed grammars," *Informatica (Slovenia)*, vol. 24, no. 3, 2000.
7. C. Bürger, "Reference Attribute Grammar Controlled Graph Rewriting: Motivation & Overview," in *SLE*. ACM, 2015.
8. F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge university press, 1999.
9. C. Bürger, J. Mey, R. Schöne, S. Karol, and D. Langner, "Using Reference Attribute Grammar-Controlled Rewriting for Energy Auto-Tuning," in *10th International Workshop on Models@run.time*, Ottawa, Canada, Sep. 2015.

Search Space Reduction for E/E-Architecture Partitioning

Andreas Ettner

Robert Bosch GmbH, Corporate Reasearch,
Robert-Bosch-Campus 1, 71272 Renningen, Germany
andreas.ettner@de.bosch.com

Abstract. As the design of electrical/electronic (E/E)-architectures is becoming more complex, multi-objective optimization algorithms such as evolutionary algorithms (EAs) have been proposed for generating resource optimized architectures. In this paper we extend existing approaches by excluding infeasible solutions from the search space and thereby enhance the quality and runtime behavior of the optimization.

Keywords: E/E-Architecture Partitioning, Design Space Reduction, Evolutionary Algorithm

1 Introduction

Due to the introduction of new safety and comfort functions related to advanced driver assistance, highly automated driving, and car-to-x connectivity, the number and interconnections of functions in vehicles has been growing over the last decades and is going to grow further over the next years. As a result, distributed vehicle system architectures consisting of heterogeneous computing resources become more complex and power consuming. Thereby, also the complexity of the allocation task problem is growing, which is defined as assigning functions to Electronic Control Units (ECUs) while fulfilling various design constraints, such as safety requirements and resource restrictions (see Figure 1).

In recent years, several optimization methods with the aim of supporting engineers in power and resource aware design of E/E-architectures have been proposed. While Walla [6] presented a mixed-integer linear programming (MILP) approach to optimize function partitioning with respect to energy efficiency, EAs have proven useful for concurrent optimization of multiple objectives, such as performance, cost, and reliability [1], [3], [4]. However, when increasing the number of design constraints, EAs might fail in producing feasible solutions and in converging toward a global maximum. Common approaches to overcome this problem have been compared by Moser [5] and the results showed that repairing infeasible solutions is superior to penalizing and constrain-dominance methods. Yet, the search space still contains all infeasible solutions for each of which the repair function would be invoked. In order to overcome this drawback and to decrease the amount of infeasible solutions, we present an approach to reduce the search space in advance of the optimization run. Thereby, we enhance the quality and runtime behavior of the optimization.

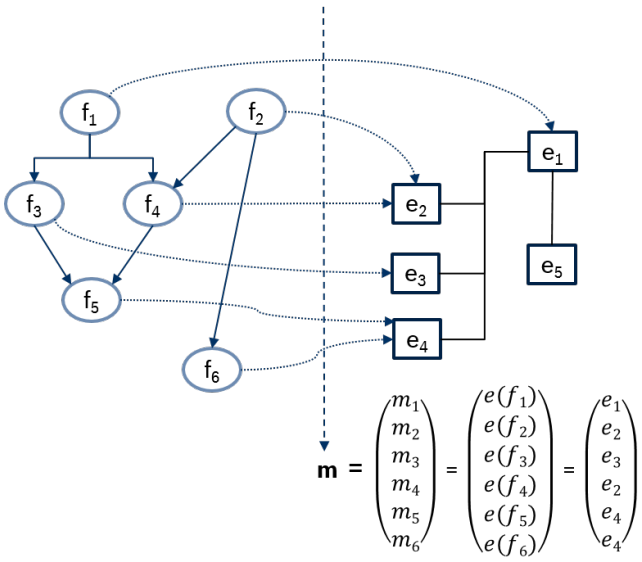


Fig. 1. Function Allocation

2 Concept

2.1 Models and Representation

Figure 1 exemplary shows the two input models to the optimization. The function model consists of a set F of functions with inter-dependencies being represented by directed edges. The architecture is modelled by a set E of ECUs connected by bus structures. Properties describe the available resources on the ECUs and the resource demands as well as functional requirements of the functions. For example, the computing requirements for each function are represented by r_f and the available computing resources for each ECU by r_e, respectively.

The allocation of vehicle functions to the ECUs is represented by a mapping vector m indicating for each function f_i the corresponding ECU identifier e_j. By default, each function can be allocated to each ECU, such that the set of all possible ECUs for f_i is $M = \{e_1, ..., e_J\}$ and the number of possible solutions is $S = |E|^{|F|}$.

2.2 Objective Functions and Constraints

In our approach, we optimize three objective functions: network communication, safety, and the collection of functions with certain properties on a minimum number of ECUs. As the search space reduction is independent of the objective functions, we will concentrate on the constraints in the following:

- Location constraints either define a set of ECUs $C_{I,i}$ - where function f_i will be mapped to one element of this set - or exclude a set of ECUs $C_{E,i}$.

- Colocation constraints either forbid two functions to be allocated to the same ECU ($C_{ban} : e(f_i) \neq e(f_j)$) or force two functions to reside on the same ECU ($C_{force} : e(f_i) = e(f_j)$).
- Some functions demand certain components or functional requirements reqf, which have to be provided by the corresponding ECU (req_e). Therefore, function f_i can only be allocated to one ECU of the set $C_{req,i} = \{e | req_e \geq req_f\}$.
- Some resources, such as the memory and computing units, are shared among all functions allocated to the ECU. Therefore, $r_{e,j} \geq \sum r_{f,i}$ with $(e(f_i) = e_j)$ has to hold for each ECU.

2.3 Genetic Algorithm

For the optimization, we use the NSGA-II algorithm presented by Deb [2]. Its structure is shown in Figure 2. During initialization, a population of solutions is generated by assigning one element of set M to each of the functions resulting in one mapping vector for each solution. Afterwards, the population iteratively improves by creating new solutions, evaluating these solutions with regard to the constraints and the objective functions (see Fig. 3), and finally selecting the best solutions for the next generations based on Pareto-optimality.

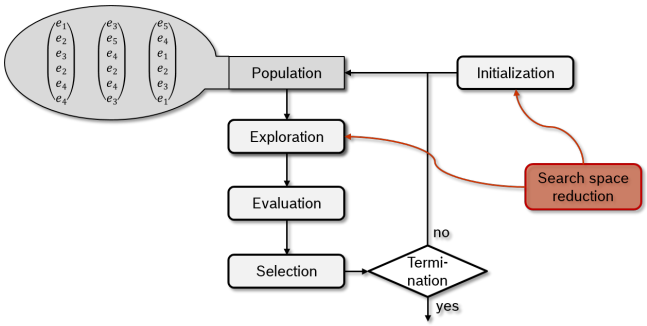


Fig. 2. Structure of Evolutionary Algorithm

New solutions are generated by variation operators such as mutation, which assigns an element of set M to a random number of functions. Thereby, a huge amount of solutions violating the constraints defined in chapter 2.2 might be generated that would have to be either repaired or discarded. As an approach to reduce this number, we reduce the sets M_i for each function f_i in advance of the optimization by analyzing the design constraints and use those sets during initialization and mutation to generate new individuals.

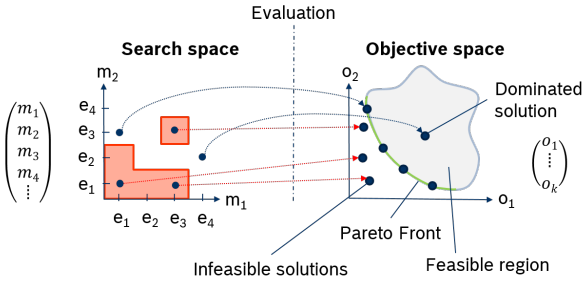


Fig. 3. Evaluation of individuals

2.4 Search Space Reduction (SSR)

Figure 4 presents the SSR for each function. In a first step, the sets Mi are initialized with either the ECUs defined in $C_{I,i}$ (1a) or, in case $C_{I,i}$ is not defined, with all ECUs from M (1b). Afterwards, M_i is reduced by $C_{E,i}$ (2), by the ECUs that do not fulfil the constraints on functional requirements (3), and by the ECUs that do not provide sufficient resources for function f_i (4). Step (5) ensures that the sets M_i and M_j of two functions to be forced together contain the same ECU elements. Finally, if functions to be banned from f_i are already allocated to a certain ECU, this ECU is removed from M_i (6).

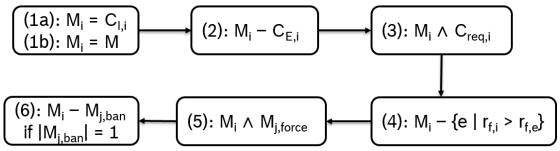


Fig. 4. Reduction sequence

3 CASE STUDY AND RESULTS

Table 1 and Table 2 exemplary show the property values for the optimization problem presented in Figure 1.

	f_1	f_2	f_3	f_4	f_5	f_6
r_f	6	2	3	1	2	3
f_req_f	8	0	4	4	2	1

Table 1. Function requirements

	e_1	e_2	e_3	e_4	e_5
r_e	10	10	5	10	10
f_req_e	8	4	8	2	1

Table 2. ECU properties

original	(1)	(2)	(3)	(4)	(5)	(6)
15,625	9,375	7,500	810	405	243	162
100 %	60 %	48 %	5.2 %	2.6 %	1.6 %	1 %

Table 3. Size of search space after each reduction step

Furthermore, let $C_{I,2} = \{e_1, e_2, e_3\}$, $C_{E,5} = \{e_1\}$, and consider $force(f_5, f_6)$ and $ban(f_1, f_2)$. Applying SSR to the optimization problem results in $M_1 = \{e_1\}$, $M_2 = \{e_2, e_3\}$, $M_3 = \{e_1, e_2, e_3\}$, $M_4 = \{e_1, e_2, e_3\}$, $M_5 = \{e_2, e_3, e_4\}$ and $M_6 = \{e_2, e_3, e_4\}$. As shown in Table 3, the search space could be pruned to about 1 % of its original size. This has also an impact on the number of generations for finding the optimal solutions. With a population size of 10, we commonly find the five Pareto-points after 15 generations, whereas an EA with repair mechanism needs 150 generation and an EA without repair function and mapping sets more than 5000 generations. For a larger use case with 32 functions, 10 ECUs, location constraints on 28 functions, and a population size of 50, Figure 5 shows the mean hypervolume values over 25 optimization runs. Whereas the standalone NSGA-II finds first feasible solutions after 60 generations and then slowly increases, the NSGA-II with SSR converges after 140 generations.

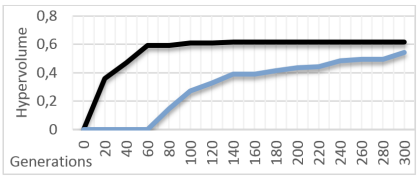


Fig. 5. Hypervolume with SSR (black) and without (blue)

References

1. Blickle, Tobias, Jrgen Teich, and Lothar Thiele. "System-level synthesis using evolutionary algorithms." Design Automation for Embedded Systems 3.1 (1998): 23-58.
2. Deb, Kalyanmoy, et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." Evolutionary Computation, IEEE Transactions on 6.2 (2002): 182-197.
3. Hardung, Bernd. Optimisation of the allocation of functions in vehicle networks. Shaker, 2006.
4. Moritz, Ralph, Tamara Ulrich, and Lothar Thiele. "Evolutionary exploration of e/e-architectures in automotive design." Operations Research Proceedings 2011. Springer Berlin Heidelberg, 2012. 361-366.
5. Moser, Irene, and Sanaz Mostaghim. "The automotive deployment problem: A practical application for constrained multiobjective evolutionary optimisation." Evolutionary Computation (CEC), 2010 IEEE Congress on. IEEE, 2010.
6. Walla, Gregor, et al. "An automotive specific MILP model targeting power-aware function partitioning." Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on. IEEE, 2014

An Evaluation of Autotuning Techniques for the Compiler Optimization Problems

Amir Hossein Ashouri, Gianluca Palermo and Cristina Silvano

Politecnico di Milano,
Milan, Italy
{amirhossein.ashouri,ginaluca.palermo,cristina.silvano}@polimi.it

Abstract. Diversity of today's architectures have forced programmers and compiler researchers to port their application across many different platforms. Compiler auto-tuning itself plays a major role within that process as it has certain levels of complexities that the standard optimization levels fail to bring the best results due to their average performance output. To address the problem, different optimization techniques has been used for traversing, pruning the huge space, adaptability and portability. In this paper, we evaluate our different autotuning approaches including the use of Design Space Exploration (DSE) techniques and machine learning to further tackle the both problems of selecting and the phase-ordering of the compiler optimizations. It has been experimentally demonstrated that using these techniques have positive effects on the performance metrics of the applications under analysis and can bring up to 60% performance improvement with respect to standard optimization levels (e.g. -O2 and -O3) on the selection problem and up to 4% w.r.t. to LLVM's standard optimization on the phase-ordering problem.

1 Introduction

Conventional software applications are first developed in the desired high-level source-code (e.g. C, C++) and then are passed through the compilation phase to build the executable. The later phase includes compiler optimization process in which the target metrics such as execution time, code-size, power, etc are optimized depending on the desired scenario. Compiler optimizations are playing an important role to transform the source-code to an optimized variation. Usually, open-source/industrial compiler platforms are coming off-the-shelf with some standard optimization levels (e.g. `-O1, -O2, -O3` or `-Os`) to bring the average-good results for conventional platforms. However, quite often they fail to bring the optimal results for specific applications, architectures and platforms. In the short paper, two different techniques for compiler auto-tuning , namely, DSE and Machine Learning based techniques have been proposed to accommodate and address the problem of selecting the best compiler optimization for a given application.

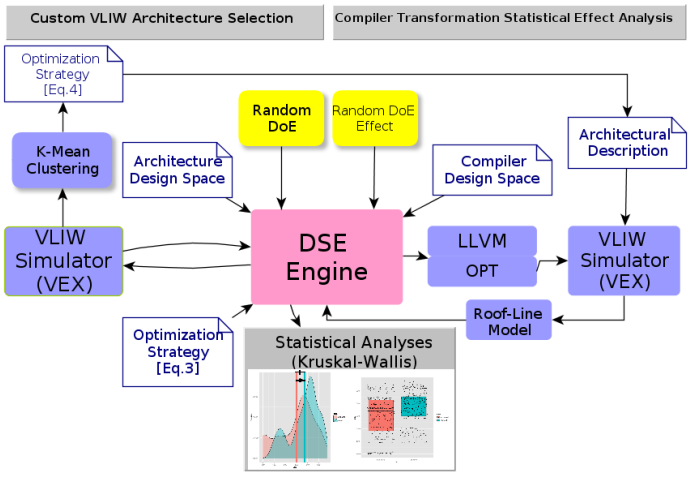
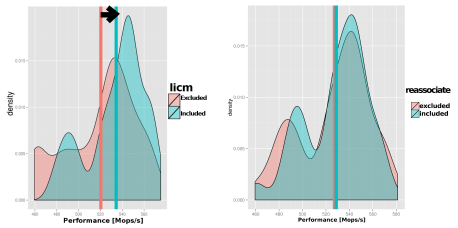


Fig. 1: Approach (I): DSE Proposed Methodology

2 Approach (I): DSE Approach

Design Space Exploration (DSE) refers to the activity of exploring the design parameters alternatives before the actual design. It deals with pruning and exploring the design space efficiently. The proposed work targets the exploration of compiler options parameters, in order to automatically explore the design space and analyze the compiler-architecture co-design. We experimentally assessed the proposed methodology in Very-long-Instruction-Word (VLIW) architecture by applying random Design of Experiment (DoE) and an automatic tool-chain including our Multi-Objective System Tuner tool (MOST), a wrapper and a compiler/simulator; namely, LLVM and VLIW-EXample (VEX). It enables to automatically explore, optimize and analyze the optimizations by using several standard benchmarks for both high-end embedded and signal processing applications [1]. Analytically, we show that the adoption of the specific methodology either in a cross-architecture and/or cross-application manner, can deliver significant application specific insights thus enabling the designer to guide through decisions regarding the architecture and the compilation optimization strategy [2]. Figure 1 represents the proposed methodology for compiler co-exploration with DSE. The work-flow starts by inferring the *Pareto-optimal* architectural design space and then feeding the found architectural properties to the compiler framework. Statistical analyses will be applied at the end to assess the correlation between utilizing the certain compiler options and the observed performance metrics. Figure 2 is showing different distributions derived by applying the proposed DSE.



(a) (b)

Fig. 2: Visualization of (a) *licm*'s significant positive effect, (b) *reassociate*'s no significant effect

3 Approach (II): Machine Learning Approach

Diversity in applications and architecture, simply makes it barely possible to manually optimize and port the source-codes for each application/architecture. *Random Iterative Compilation* fails to efficiently bring the optimal results due to its high demand on time and number of iterations. In order to improve the portability of compiler optimization with respect to the handcrafted approaches, machine learning has been used to address both the *selection of compiler optimization options* and *phase-ordering problem* [3] to predict the right optimization to be applied given an unseen application [4].

3.1 The Problem of Selecting the Right Set of Compiler Optimizations

Addressing the issue on the second approach, we propose a Machine Learning based autotuning framework that maximizes the performance of a target application. COBAYN: Compiler Autotuning Framework Using Bayesian Networks, starts by applying statistical methodology with Bayesian Networks to infer the probability distribution of the compiler optimizations to be enabled to achieve the best performance. We start to drive the iterative compilation process by sampling from the probability distribution. Likewise most machine learning approaches, here we use a couple of sets of training applications to learn the statistical relations between application features and the compiler optimizations. Given a new unseen application, its features are fed into the machine leaning algorithm as *evidence* on the distribution. This evidence imposes a bias on the distribution. Since compiler optimizations are correlated with the software features, we can redo the process of sampling for the new target application. Figure 3 demonstrates the second proposed approach that is assessed on an embedded *ARM* device with GCC compiler. The obtained probability distribution is indeed application-specific and effectively exploits the use of iterative compilation process as it only drives with the most promising compiler optimizations [5, 6]. Figure 4, represents the result of our proposed methodology, COBAYN, against GCC's standard optimization levels `-O2` and `-O3` when using *cBench* suite. It represents significant speedup factor over the the experimentally tested applications with the average 56% and 47% improvement, respectively against `-O2` and

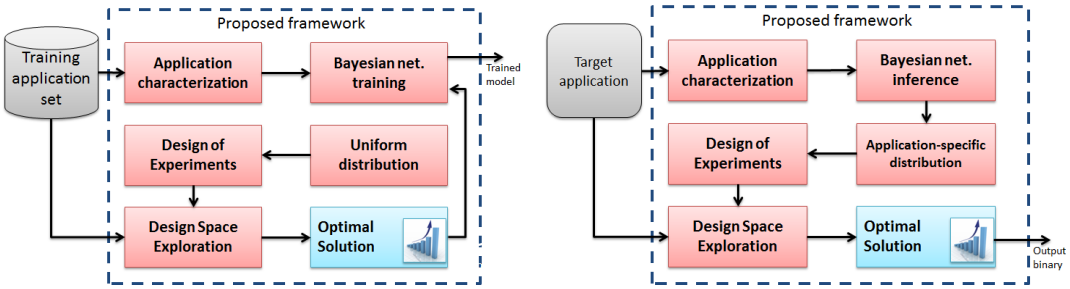


Fig. 3: Approach (II): Overview of the proposed M.L Methodology a) training phase b) predicting phase

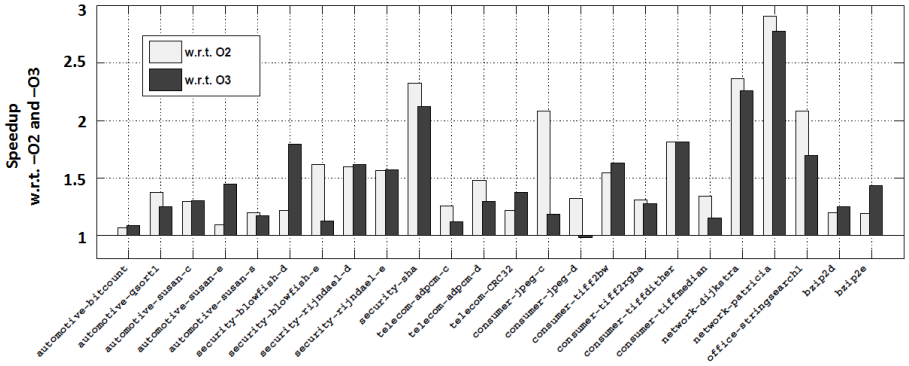


Fig. 4: Performance improvement of our Bayesian Networks w.r.t -O2 and -O3

`-O3`. COBAYN, led to reach a factor of 3× exploration speedup compared with the *random iterative compilation* having a fixed number of predictions.

3.2 The Phase-ordering Problem

when taking into considerations the order of the appearance of the compiler optimizations, the so-called *phase-ordering* problem comes into play. The space gets enormously bigger and simple classic supervised techniques are not able to tackle accurate models for prediction. Addressing the phase-ordering problem, we propose an *intermediate speedup* predictor that is able to predict the current optimization to be applied given the state of the code being optimized. We used predictive models and dynamic software characterization to construct the application specific models in cross-validation manner. In order to speedup the exploration on the space, we defined two traversing heuristics , depicted in Figure 5, that use Depth First Search (DFS) and exhaustive search within the prediction space. The proposed approach reaches up to 4% speedup w.r.t LLVM's default compilation performance [3].

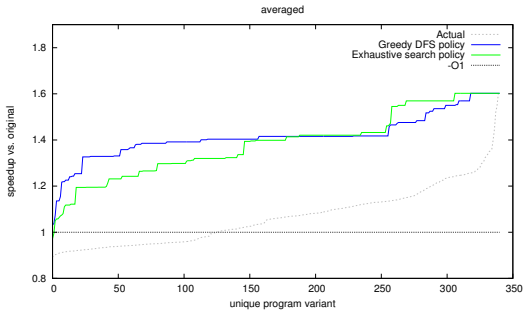


Fig. 5: Performance improvement of our proposed speedup prediction models against LLVM

4 Conclusion

This paper presents two main approaches for the compiler autotuning problem using DSE and machine learning. The experimental results is shown speedup on the performance metrics while classifying the effective compiler optimizations derived by DSE approach and 40% - 60% speedup with against GCC's -O2 and -O3 on an ARM embedded-board using COBAYN.

References

1. A. H. Ashouri, "Design space exploration methodology for compiler parameters in vliw processors," 2012, http://hdl.handle.net/10589/72083.
2. A. H. Ashouri, V. Zaccaria, S. Xydis, G. Palermo, and C. Silvano, "A framework for compiler level statistical analysis over customized vliw architecture," in *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*. IEEE, 2013, pp. 124–129.
3. A. H. Ashouri, A. Bignoli, G. Palermo, and C. Silvano, "Predictive modeling methodology for compiler phase-ordering," in *Proceedings of 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 5th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*. ACM, 2016.
4. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2006, pp. 295–305.
5. A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, p. 21, 2016.
6. A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, "A bayesian network approach for compiler auto-tuning for embedded processors," in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*. IEEE, 2014, pp. 90–97.

A Combined Fault Detection and Discrimination Strategy for Resource-Sensitive Platforms

Richard McWilliam, Philipp Schiefer, and Alan Purvis

School of Engineering and Computing Sciences, Science Laboratories,
Durham University, Durham, DH1 3LE, United Kingdom
{r.p.mcwilliam,philipp.schiefer,alan.purvis}@durham.ac.uk
http://www.dur.ac.uk/ces

Abstract. This paper presents a combined fault detection and discrimination strategy for CMOS logic incorporating active resource mitigation and monitoring. The approach is demonstrated for a NOR gate using a dual redundant gate design with selective mitigation and analogue or digital detection. The potential benefits of the approach are discussed with respect to resource awareness and management within fine-grained logic.

Keywords: Self-repair, fault detection, redundancy.

1 Introduction

Fault detection and mitigation within CMOS logic structures is a long-standing challenge that is seeing new a emphasis for nanoscale and printable electronics. The possibility for intrinsic resource awareness and management without obfuscating management at higher design levels is an attractive proposition but requires new gate and transistor level strategies. This paper presents ongoing work into a combined fine-grained redundancy and active mitigation approach with minimal resource overhead that enables selective fault detection, masking and discrimination close to the point of fault manifestation.

1.1 Existing Methods

On-line fault strategies have been discussed at length for future nanoscale electronics where massive redundancy concepts become feasible [1]. However, resource-sensitive platforms typically involve more conservative duplicate gate and/or interconnect structures combined with majority signal generation in order to mask faults and prevent their manifestation at critical outputs. Practical examples involving triple and quad redundancy are illustrated in Fig. 1a-b. Combined logic interleaving and quad-transistor structures have also been investigated [2]. While the use of regular cell structures is attractive, typical methods incur between 3–8 times resource overhead and do not achieve fault detection or discrimination. It could be argued that fault detection triggers may be generated within quad-transistor majority logic but determination of the specific fault location and its

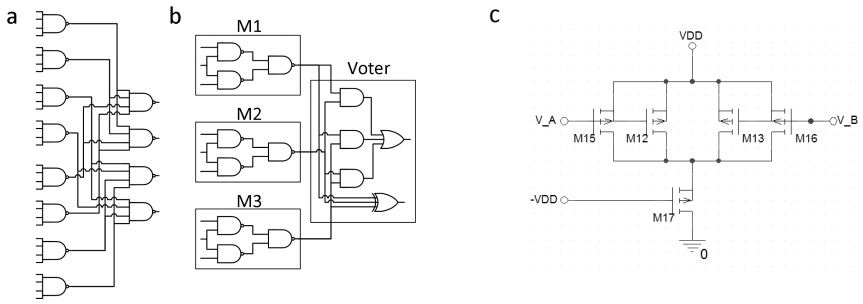


Fig. 1. Traditional fault-tolerant designs. a) Quadded logic. b) Triple-modular redundancy with voter. c) pseudo-CMOS with redundancy.

type becomes abstracted by the internal process of converting critical faults to sub-critical faults.

Fine-grained fault tolerant strategies are beginning to feature in future nano-scale CMOS logic design with the principal aim of combating manufacturing defects. This includes psueo-CMOS redundancy, a simple example of which is illustrated in Fig. 1c. Another approach is reported in [3] wherein defects present in either N-type or P-type networks invokes switched active pull up or pull-down loads. In this case, however, defect detection is not part of the repair method and instead would be provided by additional built-in self-test (BIST) logic and possibly external test equipment. Hard-fault mitigation approaches have been proposed that are based on active switching matrices [4]. However, self-detection is once again not included as a part of the strategy.

Field-programmable gate arrays (FPGA) provide flexible platforms featuring configurable cellular architectures that support full or partial configuration. Since their total resource utilisation rarely approaches 100%, there are opportunities to provision redundant resources for fault mitigation. Even so, it is not yet clear how spare resources may be reallocated to support online fault detection and discrimination without resorting to external supervisory hardware/software as typified in [5]. While solutions based on custom programmable architectures have been proposed that aim to address this limitationby enabling dynamic resource allocation [6] , fault detection is still achieved through data error detection and correction (EDC) hardware that is abstracted from the hardware fault.

2 Proposed Method

The proposed strategy relies upon an alternative method referred to here as *Stuck-At Fault Resilient* (SAFR) design, wherein fixed dual redundancy is combined with a fault triggering mechanism [7]. An example logic NAND gate implemented by the SAFR approach in comparison to the standard NAND gate design is shown in Fig. 2, where dual redundancy is employed within the P- and N-type networks. This is in contrast to quad redundant strategies (Fig. 2c).

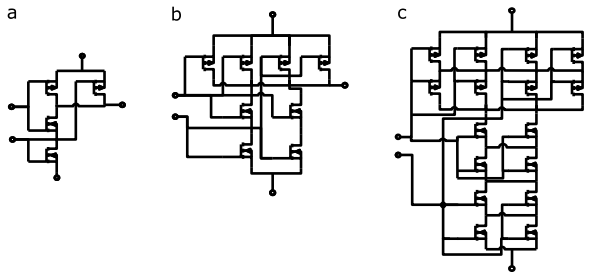


Fig. 2. Stuck-at fault resilient design. a) traditional NAND. b) proposed design. c) full quad-transistor design.

2.1 Detection Strategy

The dual redundancy strategy permits masking of any single stuck-off fault and selective fault triggers for stuck-on faults depending upon the state of the inputs. Of particular note is the fact that fault discrimination is not retained when higher redundancy factors are used i.e., triple- and quad-transistors. Hence, a resource trade-off between fault masking capacity and fault identification is present in this approach.

2.2 Discrimination and Mitigation

Selective fault masking allows for the detection of stuck-on faults considered to be critical due to potential high current flow between VDD and GND. Examination of the gate output response under fault condition, summarised in Table 1, shows that at there is at least one input combination that generates current flow between VDD and VSS for every single stuck-on fault. This may be exploited to achieve discrimination of fault type by monitoring current imbalance in the CMOS network or else periodic exercising of the gate inputs via digital test. The P- and N-networks are combined with the switching network for a NOR gate implementation are shown in Fig. 3, which includes weak active pull-up/down loads typically used for defect repair [3], but which are used here for selective online fault discrimination.

3 Resource Awareness and Management

Resource considerations will be important for emerging printable and nanoscale electronics due to their differing densities and scope for building redundancy structures based upon multi-gate and/or sub-gate nano-structures. Resource management extending to the fine-grained levels should be explored for both defect tolerance and hard-fault mitigation. Combining the above approach with weak active pull-up/down loads creates an efficient active mitigation mechanism that, when further combined with dual redundancy within the P- and

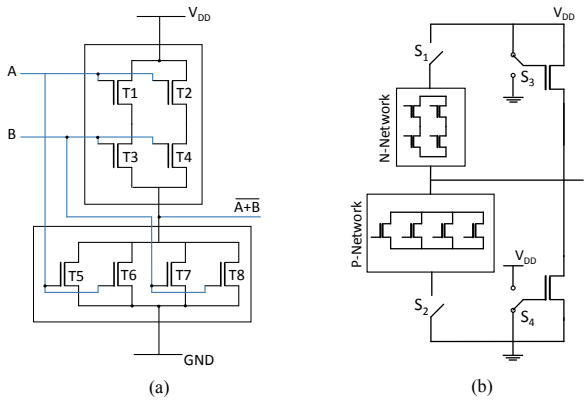


Fig. 3. Gate design strategy. (a) Example of redundancy scheme for NOR gate employing P- and N-type networks. (b) Potential implementation for active fault mitigation according to [3].

Table 1. Stuck-High Fault Response of CMOS Network

Input	Stuck-on fault location[a]							
AB	T1	T2	T3	T4	T5	T6	T7	T8
00	1	1	1	1	X	X	X	X
01	0	0	X	X	0	0	0	0
10	X	X	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0

[a] Output error denoted by 'X'

N-networks, creates further resource awareness opportunities in the presence of faults. Once a fault has been detected, partial isolation proceeds by switching to pseudo-NMOS or PMOS mode wherein the nature of the fault may be further characterised. For example, assuming a stuck-at high fault occurring within the P-network (Transistors T1-T4 in Fig. 3a), the location of the fault is not known *a-priori*. The circuit may first be switched to pseudo-NMOS mode (setting switches S1 and S3 in Fig. 3b) and, due to the complimentary nature of the design, a second analogue/digital test will would reveal the same fault behaviour summarised in Table 1. However, depending on the value of the weak pull-down resistance of transistor T9, the digital test may pass without error and the adapted circuit may continue to be used in a degraded state. Alternatively, the circuit may be switched into pseudo-PMOS mode (switches S2 and S4) whereupon the error no longer persists. Hence the state of the P- and N-networks may be individually ascertained. The reverse situation of a fault occurring within the N-network would proceed in identical fashion as described above. At all times stuck-at low fault events are intrinsically masked.

An further extension of resource awareness concerns continual resource monitoring in the presence of intermittent faults. For the above case of the pseudo-PMOS configuration being activated in response to a stuck-high fault within the P-network, a further option would be to periodically switch to the pseudo-NMOS configuration and check the P-network response to determine whether the fault persists. This serves two functions: first, intermittent faults may be handled in a graceful manner and with specific knowledge of their locality. Second, disappearance of the fault allows for restoration of the full CMOS network and non-degraded performance.

4 Conclusions

Fault detection and discrimination remains a fundamental challenge in resource management for integrated fault mitigation. The proposed dual redundancy SAFR method achieves a combination of fault discrimination between stuck-high/stuck-low fault events and selective masking, thus reserving active mitigation for stuck-high faults. Fine-grained resource mitigation proceeds by combining redundancy with weak pull-up/down networks. Ongoing work is investigating further logic gate configurations and functional logic built from such gates.

Acknowledgement

This work was supported by the UK EPSRC Centre for Innovative Manufacturing in Through-life Engineering Services (EP/I033246/1).

References

1. J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.
2. J. Han, E. Leung, L. Liu, and F. Lombardi, "A Fault-Tolerant Technique Using Quadded Logic and Quadded Transistors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–1, 2014.
3. M. Ashouei, A. Singh, and A. Chatterjee, "Reconfiguring CMOS as Pseudo N/PMOS for Defect Tolerance in Nano-Scale CMOS," in *21st International Conference on VLSI Design, 2008. VLSID 2008*, 2008, pp. 27–32.
4. R. Kothe, H. Vierhaus, T. Coym, W. Vermeiren, and B. Straube, "Embedded Self Repair by Transistor and Gate Level Reconfiguration," in *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, 2006, pp. 208 –213.
5. J. Emmert, C. Stroud, and M. Abramovici, "Online Fault Tolerance for FPGA Logic Blocks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 2, pp. 216–226, Feb. 2007.
6. P. Bremner, Y. Liu, M. Samie, G. Dragffy, A. G. Pipe, G. Tempesti, J. Timmis, and A. M. Tyrrell, "SABRE: a bio-inspired fault-tolerant electronic architecture," *Bioinspir. Biomim.*, vol. 8, no. 1, p. 016003, Mar. 2013.
7. P. Schiefer, R. McWilliam, and A. Purvis, "Fault Tolerant Quadded Logic Cell Structure with Built-in Adaptive Time Redundancy," *Procedia CIRP*, vol. 22, pp. 127–131, 2014.

Auto-tuning Fault Tolerance Technique for DSP-Based Circuits in Transportation Systems

Ihsen Alouani, Smail Niar, Yassin El-Hillali, and Atika Rivenq

[1] I. Alouani and S. Niar LAMIH lab
University of Valenciennes
France
firstname.name@univ-valenciennes.fr
[2] Y. El-Hillali and A. Rivenq IEMN lab
University of Valenciennes
France
firstname.name@univ-valenciennes.fr

Abstract. As new technologies use a reduced transistor size to improve performance, circuits are becoming remarkably sensitive to soft errors that become a serious threat for critical applications reliability. Most of the existing reliability enhancing techniques lead to costly hardware. The masking phenomenon is fundamental to accurately estimating soft error rates (SER). The first contribution of this paper is a new cross-layer model for input-dependent Single Event Transient (SET) masking mechanisms combining Transistor Level Masking (TLM) and System Level Masking (SLM). We, secondly, use this model to build an auto-tuning fault tolerant circuit dedicated to obstacle detection systems in railway transportation. Based on our input-dependent masking model, the proposed architecture evaluates the effective circuit's vulnerability at runtime and accordingly adapts the reliability boosting strategy, leading to a reliable circuit with optimized overheads. When compared to the Triple Modular Redundancy, our technique reduces the number of FPGA LUTs (resp. DSP slices) by up to 45% (resp. 33%).

1 Introduction and Related Works

Technology scaling has enabled fabulous improvements in embedded systems performance. Nevertheless, as transistor gate dimensions decrease to the nanometer scale, electronic systems become highly susceptible to environmental-factors-induced errors. Soft errors are caused by particle strikes that temporarily corrupt data stored in memory cells, or change the state of internal combinational circuit nodes. The masking phenomenon is one of the most important fundamentals involved in failure rates estimation within semiconductor circuits. In the existing works, three masking mechanisms preventing combinational circuits from soft errors have been considered [1]: Logical Masking, Electrical Masking and Latching-Window Masking. The most widely used reliability enhancement techniques in the literature are: *spatial* redundancy and *temporal* redundancy.

In this paper, we present ARDAS for *Auto-tuning Redundancy in DSP-based Architectures for Soft errors resiliency*, an architecture that uses auto-tuning redundancy of DSP blocks to protect the vulnerable circuit parts instead of protecting the whole circuit. The vulnerability analysis is performed through design-time simulations that implement the proposed masking models (TLM and SLM).

2 TLM: Transistor-Level Masking Mechanism

TLM occurrence is led by the affected transistor locality within the struck gate as well as the input combination during the transient event. In fact, the particle strike temporarily corrupts combinational elements by affecting the state of the hit transistor. However, the event can be simply unnoticed at the output if the transistor behavior corruption doesn't affect the overall state of pull-up/pull-down network. Let be D_i a binary variable set to 1 if the error due to a particle strike hitting a transistor Q_i is masked by a TLM mechanism. P_i is the probability that Q_i is the hit transistor within the struck gate by the particle. Hence, the probability that the error resulting from a radiation strike in gate j is masked for a given input combination in gate j is then expressed by: $P_{TLM}(j) = \sum_{i=1}^{N_j}(P_i \times D_i)$. For simplicity, we assume the equiprobability of gates' transistors to be hit by a particle. Let N_m be the number of cases the error is masked for a given input combination. Hence, $P_{TLM}(j) = \frac{N_m}{N_j}$.

The probability of soft error masking in the output bit S_i of a combinatorial circuit for given input signals is:

$$P^i_{masking} = \sum_{j=1}^{n} W_j \cdot (P_{TLM}(j) + (1 - P_{TLM}(j)) \cdot D_{ij}) \quad (1)$$

Where n is the number of gates in the circuit, $P_{TLM}(j)$ is the probability of TLM at gate j and W_j is the weight assigned to gate j, expressed as the number of the gate's transistors divided by the total number of transistors in the circuit. Finally, D_{ij} is a binary variable set to 1 if the error at gate j does not propagate to output S_i and to 0 otherwise.

3 SLM: System-Level Masking Mechanism

In a threshold-based system, the comparison of the intermediate result with a beforehand fixed threshold gives the overall system decision. A transient error in the intermediate result may keep the overall system decision unchanged depending on the detection threshold value.

We consider a widely used signal processing element in detection/recognition applications, namely a correlator. We built a simulation tool that tracks the propagation of event-induced errors happening within the correlator nodes and evaluated their impact on obstacle detection accuracy. The correlator is implemented

using DSP48E1 slices [2]. A soft error is modeled by injecting a bit flip in a node (i, j) corresponding to the output bit i of the DSP_j. Hence, the system behavior can be monitored under fault injection through *System Failures* (SFs) detection. A SF corresponds either to a "False Alarm", or a "No Alarm". To identify SFs, we introduce the variable δ_{ij} that is expressed by: $\delta_{ij} = ({C_{ij}}^* - Y_0) \cdot (C - Y_0)$, where ${C_{ij}}^*$ is the correlation result under fault injection in node (i, j), C is the error-free result and Y_0 is the correlation threshold. A SF occurs when $\delta_{ij} < 0$. However, if $\delta_{ij} \geq 0$ we have a System Level Masking (SLM).

4 ARDAS: Proposed Approach

We define V_j, the vulnerability of a DSP_j by:

$$V_j = \frac{\sum_{i=1}^{N_j} \eta_{ij} \cdot (1 - P_{ij})}{N_j} \tag{2}$$

Where: N_j is the number of output bits of DSP_j, P_{ij} is the probability of TLM relative to bit i of DSP_j and η_{ij} is a variable set to 0 if a fault at node (i, j) is masked by SLM, i.e. $\delta_{ij} \geq 0$ and is equal to 1 otherwise. We localize vulnerable DSPs as those with $V_j > V_0$ and define α_j as follows: $\alpha_j = 0$ if $V_j > V_0$ and $\alpha_j = 1$ if $V_j \leq V_0$. As $[V_j]$ vector depends on the applied input signals, the redundancy distribution corresponding to the vulnerability map has to be dynamically tunable and self adaptive. The main idea is to judiciously use the redundant DSP slices to carry out an auto-tuning partial TMR instead of a full TMR. The system adapts the redundancy to the actual vulnerability map of the circuit using the circuit's $[\alpha_j], \forall j \in [1; N_{dsp}]$.

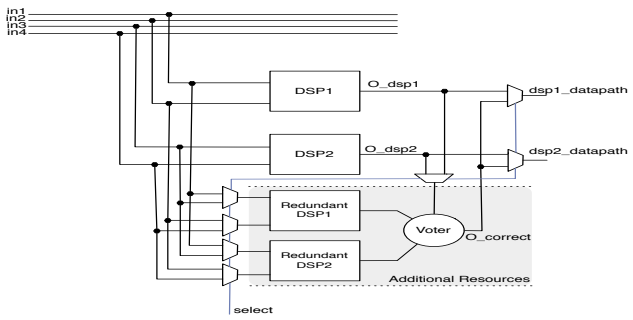


Fig. 1. An illustrative circuit of the auto-tuning redundancy used in ARDAS

The reconfiguration process used to change the redundancy mapping at run-time is taken from our previous work [3] and an example is illustrated in Figure 1. The circuit mapping is configured by a single control word according to a redundancy map obtained offline through design-time simulations.

5 Experimental Results

We compare the SER of ARDAS-protected to a TMR-protected correlation circuit. As the reliability level is tuned via V_0, Figure 2 represents the normalized SER and the number of used DSP slices in ARDAS in terms of the tolerated DSP vulnerability threshold. As seen in Figure 2, the reliability level provided by ARDAS is comparable to TMR reliability level with lower HW resource utilization.

Table 1. Resource utilization, power and maximum frequency.

	Original	TMR	ARDAS V_0=0.55	ARDAS V_0=0.7	DTR
DSPs	79	237	189	159	79
LUTs	0	1798	1619	1207	1413
Pw(mW)	430	691	542	533	459
Max freq (MHz)	422	247	347	347	410

In addition to the reliability, we investigate the impact of ARDAS on power consumption, resource utilization and the maximum clock frequency of each circuit for two vulnerability threshold values: 0.55 and 0.7. The circuit is synthesized for a Xilinx Virtex 7 board. The power consumption is estimated using the Xilinx XPower Analyser tool. Table 1 shows that our architecture reduces the reliability cost in terms of resource utilization, power and performance. In fact, ARDAS decreases the number of used LUTs by 10% for $V_0 = 0.55$ and by 32% for $V_0 = 0.7$ compared to TMR. On the other hand, while using TMR slows

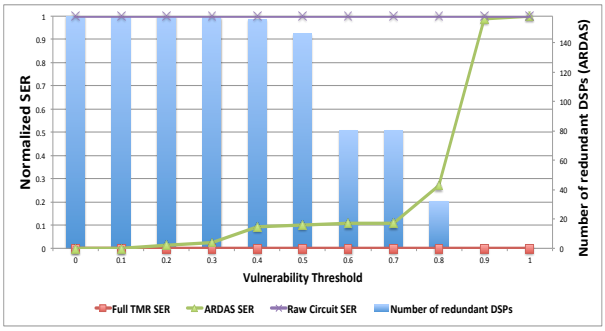


Fig. 2. Normalized SER (left axis), used DSP resources vs V_0 (right axis)

down the circuit frequency by 42%, ARDAS performance penalty is less than 18% compared to the unprotected circuit.

6 Conclusion

In this paper, a self adaptive reliability approach is proposed to cope with the increasing error rates in new technologies with the lowest possible overheads. ARDAS relies on an auto-tuning redundancy architecture to protect the vulnerable parts of the system rather than the whole circuit. Due to its quick reconfigurability, ARDAS offers high reliability with reduced overheads. Moreover, it allows designers to choose the desired reliability level depending on the application requirements and its criticality.

References

1. P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Tran onNuclear Science*, June 2003.
2. 7 series dsp48e1 slice user guide. [Online]. Available: www.xilinx.com/support/documentation/user-guides/ug479-7Series-DSP48E1.pdf
3. I. Alouani, M. A. R. Saghir, and S. Niar, *Reconfigurable Computing: Architectures, Tools, and Applications: 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings*, ch. ARABICA: A Reconfigurable Arithmetic Block for ISA Customization, pp. 248–253.

Automatic Pruning of Autotuning Parameter Space for OpenCL Applications

Ahmet Erdem, Gianluca Palermo[6], and Cristina Silvano[6]

Department of Electronics, Information and Bioengineering
Politecnico di Milano

Abstract. OpenCL standard reaches more wider audience due to increasing the number of devices supporting it. This situation puts developers who want performance on large range of platforms in a difficult position. To solve this problem, autotuning frameworks are deployed. But the problem of design exploration space is seriously large because of OpenCL parameters. In this work, we introduce an approach which uses constraint programming to prune the design space before employing intelligent or exhaustive techniques to explore.

1 Introduction

The recent advances in computer architecture made heterogeneous computer systems available to not only data centers and supercomputers but also to commercial personal computers. Especially, with the advent of AMD APUs and Intel CPUs which include integrated GPUs, the heterogeneity of modern machines has increased. Furthermore, enabling discrete GPUs for general purpose computing has added another type of computation device to the system. While each system has provided different granularity of parallelism which needs to be properly exploited, the communication between various computation units must also be handled according to the needs of application as well.

Open Computing Language (OpenCL) which is maintained by Khronos consortium [3] is an open standard for developing parallel applications on heterogeneous systems by abstracting the underlying compute machine. OpenCL adopts data parallel approach by describing the parallel computations as a group of *work-items*, called *work-groups*. This hierarchical parallelism has been realized by launching kernel functions with a number of work-groups including a set of work-items. Kernel function describes how each work-item defines the operations that is to be carried out on a single data. Therefore the collection of work-items under all work-groups together expresses the data parallelism for an application. Although OpenCL defines the execution of the application that is portable between the devices conforming the OpenCL standard, it does not guarantee the performance to be optimal. Especially, moving applications to different types of architectures like from CPU to GPU may result significant loss of performance, this is the reason why OpenCL is not considered performance portable. On heterogeneous performance portability represents a challenging research issue.

One naive solution to performance portability is to develop separate kernel functions for each device the application is supposed to run. This solution makes development of application dramatically complicated when the system is heterogeneous, because of explicit management of multiple command queues and contexts in the presence of multiple vendors on the system.

Performance portability problem of OpenCL applications has been approached either by tuning of significant parameters described as in [6] or by introducing Domain-specific languages to annotate kernel and OpenCL code generation [2].

From another perspective, it is not always possible to access these parameters to tune if they are not being exposed by developers. The work of [1] tackles this problem by coalescing *work-groups* using compiler transforms while preserving the correctness of application.

In this work, we introduce an automation of extraction of OpenCL platform parameters and usage of the information that is gathered to aid the tuning process described in [6].

2 Proposed Methodology

The procedure of autotuning of an OpenCL application in order to get optimum performance without concerns of underlying architecture of the platform requires a set of parameters that define characteristics of the machine. In the case of OpenCL, these platform specific parameters are stated by the OpenCL standard itself. Furthermore, it is possible to gather them using the querying framework which is provided by the OpenCL standard. With these information gathered, it is possible to determine the size of the exploration space and then using intelligent methods for searching optimum design space.

Outside of platform parameters, there might be also application specific parameters that can be tightly related to platforms capabilities. An example of this situation is well-known tiled version of the matrix multiplication. Size of the tiles are considered as an application parameters and due to nature of the algorithm there is a sharing of information between work-items on the elements of the same tile. Due to OpenCL architecture design, this kind of communication requires local memory to be used. Therefore tile size is directly related to local memory usage which is a limited resource of the platforms.

There are some problems regarding with this approach; design space is larger for even simple applications, for instance, Nvidia Fermi architecture allows up to 1024 work-items for first and second dimensions and 64 work-item for the third dimension, resulting a 2^{26} different configurations already. Most of the configurations are not feasible in the sense that the kernel may not even launch or may fail during execution, due to illogical configurations parameters. Moreover these failed attempts of kernel launches do not provide any information about the sample that has been taken from design space. Hence effort and time are wasted on these ill-advised configurations.

In order to address this issue, the work in [6] presented a design space exploration flow that includes constraint programming to prune the design space

and eliminate infeasible solutions. This helps reduction of space while only using samples which makes sense within the scope of OpenCL standard. Fig. 1 demonstrates this idea simply.

Our work aims to improve the pruning phase by automating the extraction of platform specifications, to find constraints that are valid for all platforms, so that application programmer only needs to insert constraints related to application itself. For constraint programming, we used MiniZinc [4] constraint modelling language as it is used in [6]. Using OpenCL querying framework, for each OpenCL compliant device available on the machine we generated MiniZinc data files which include the following information about the device:

- maximum *work-group* size for each three dimensions.
- maximum number of total *work-group* a kernel launch may contain.
- number of compute units on the device.
- local memory size of the device.

In addition to these, a set of constraints that can be deduced from standard [3] has been used to generate platform constraint model, thus together with application constraints provided by programmer can prune the design space effectively. The generated platform constraints are as follows:

- total number of *work-groups* launched must be less than or equal to maximum *work-group* size.

$$workgroup_x * workgroup_y * workgroup_z <= max\_total\_wg \tag{1}$$

- each global work-item dimensions must be multiple of corresponding *work-group* dimension size.

$$\begin{aligned} global_x \% workgroup_x &== 0 \\ global_y \% workgroup_y &== 0 \\ global_z \% workgroup_z &== 0 \end{aligned} \tag{2}$$

- total number of work-groups should be equal or greater than number of compute units. Otherwise there will be idle compute units.

$$\begin{aligned} &global_x/workgroup_x + global_y/workgroup_y + \\ &global_z/workgroup_z >= num\_compute\_units \end{aligned} \tag{3}$$

3 Use Case Example

In order to test our approach, we used a machine with Intel i7-2630QM which is a quad-core CPU at 2.0Ghz and Nvidia GeForce GT 550M which is a mobile GPU with 96 CUDA cores. For testing purposes, tiled version of matrix multiplication is used and tile size is given as a application parameter and it is been set the

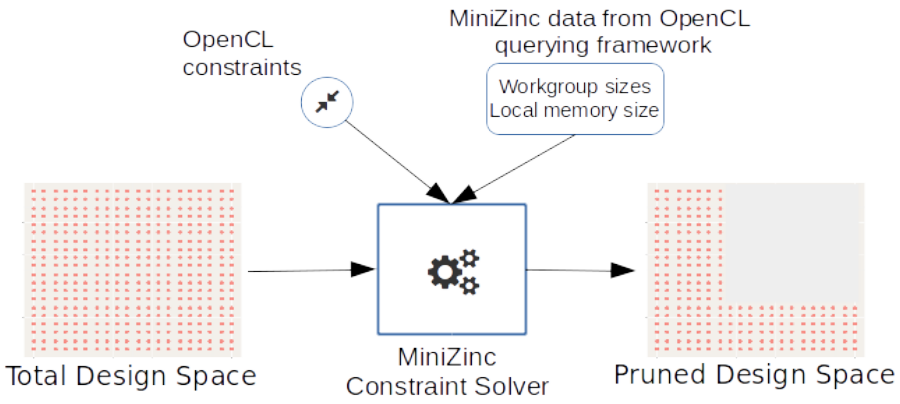


Fig. 1. Pruning of configuration design space

interval of [1 : 256]. Furthermore, we have taken into account that application global work size and set it to 4096x4096 matrix for this experiment.

Given this setting, the design space for CPU is 2^{34} and for GPU it is 2^{47}. But after the pruning of infeasible configurations, there are only 367 configurations left for CPU and 421 configurations for GPU. Considering the result of pruning for the use case given, it is even possible to search exhaustively all possible configurations left to find the optimal one. However the given example is too elementary to deduce this conclusion for broader range of applications. Hence, the next step is to introduce tools to at least one of the industry-proven benchmarks like OpenDwarfs [5], Rodinia [7] and shoc [8]. Moreover, testing with more diverse and recent hardware platforms is necessary to prove generality of the work. Additionally, besides platform specific and application specific parameters, there is a possibility of adding compiler-supported parameters like coalescing factor that has been explored in [1].

References

1. G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale. Towards transparently tackling functionality and performance issues across different opencl platforms. In *In proceedings of the Second International Symposium on Computing and Networking Across Practical Development and Theoretical Research (CANDAR 2014)*, Dec. 2014.
2. N. Chaimov, B. Norris, and A. Malony. Toward multi-target autotuning for accelerators. In *Parallel and Distributed Systems (ICPADS), 2014*, pages 534 – 541.
3. Khronos Group. The open standard for parallel programming of heterogeneous systems. [Online; Accessed: Nov. 2015].
4. MiniZinc. Medium-level constraint modelling language minizinc. [Online; Accessed: Dec. 2015].
5. OpenDwarfs. Opendwarfs. [Online; Accessed: Jan. 2016].
6. E. Paone, F. Robino, G. Palermo, V. Zaccaria, I. Sander, and C. Silvano. Customization of OpenCL applications for efficient task mapping under heterogeneous

platform constraints. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 736–741. EDA Consortium, 2015.
7. Rodinia. Rodinia:accelerating compute-intensive applications with accelerators. [Online; Accessed: Jan. 2016].
8. Shoc. Scalable heterogeneous computing (shoc). [Online; Accessed: Jan. 2016].

Application Adaptation at Runtime through Dynamic Knobs Autotuning

Davide Gadioli[1], Gianluca Palermo[1], and Cristina Silvano[1]

Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioigegneria
name.surname@polimi.it

Abstract. Several classes of applications expose a set of parameters that influence their extra-functional properties, such as the quality of the result or the size of the output. This leads the application designer to tune these parameters in order to find the configuration that produces the desired outcome.
From the architectural point of view, the trend in modern systems is to expose an high level of parallelism, often involving heterogeneous resources. To exploit the full potential of the hardware, the application designer must take into account resource-related parameters in the tuning process as well.
Since the requirements of the applications and the resources assigned to each application might change at runtime, we argue that finding a one-fit-all configuration is not a trivial operation.
For this reason we use a framework that enhances an application with an adaptation layer in order to continuously tune the parameters of the application according to the evolving situation, in a best effort fashion.

1 Introduction

One of the main tasks of an application designer is to reach the required performance on the target system. Unfortunately, the performance of an application is seldom defined by one metric, such as the execution time or its throughput. The performance is instead composed by a collection of metrics that are usually in contrast between them; for instance the time spent on elaborating the input against the quality of the result or the power consumption.

A common approach is to write an algorithm that exposes a set of parameters, also known as *dynamic knobs*[2] in literature, that influence the performance of the application, such as the number of trials in a Monte Carlo solver or the resolution of the output frame in a video encoder. The possible values of these parameters define the design space of the application and in literature are described several Design Space Exploration (DSE) techniques[4] that are able to automatically and efficiently compute the Pareto set, which represent all the optimal trade-off between the metrics of interest.

Since the application requirements may change at runtime – for instance if the platform is at first powered by a battery, then plugged in a power supply – and the system might vary the resources allocated to the application as well, we argue that is not trivial to select a priori one-fit-all configuration.

For this reason we rely on the ARGO[1] framework[1]: it is grounded on the Monitor-Analyze-Plan-Execute (MAPE) feedback loop[3] and it is able to automatically tune the application parameters according to the evolution of the system.

The main idea of the framework is to exploit design time knowledge of the application, obtained through a DSE, to select the best configuration according to the actual application requirements and the observed performance, both of them composed by a collection of metrics of interest.

ARGO is implemented as an external library to be linked against the target application. It takes autonomous decision without interacting with any other element. For these reasons we are able to minimize the intrusiveness of the integration, expressed in terms of lines of code to be changed.

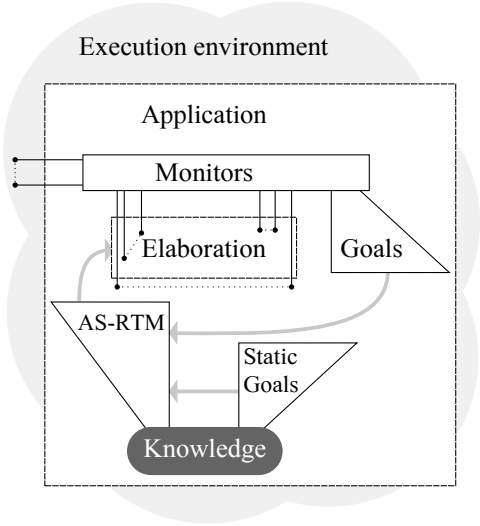


Fig. 1. The framework structure. The AS-RTM selects the best configuration according to the runtime information provided by the monitors and the design-time knowledge.

2 Framework structure

The framework follows a modular approach, as showed in Figure 1. A monitor infrastructure is used to gather insight on the actual performance of the appli-

[1] The name ARGO, has been borrowed by Greek mythology. ARGO was the ship on which Jason and the Argonauts sailed to retrieve the Golden Fleece. As that boat was a means for achieving the Golden Fleece (their goal): it aims at letting applications to reach their goals too.

cation (Monitor element). ARGOships with a monitor suite to observe the most common metrics:

- The elapsed time or the application throughput.
- The resident set size of the virtual memory that the process is using.
- The process- and system-wide CPU usage.
- Low-level metrics exposed by the widely adopted PAPI framework[6].

Moreover, to observe application-specific metrics, such as the quality-related ones, the object-oriented implementation enable the application designer to easily integrate a custom monitor, defining the methods that actually gather the new data.

On the other side, ARGOembeds the design time knowledge in the list of Operating Points (OPs), where each OP represents a configuration and the performance reached by the application using that configuration. The framework is agnostic about the technique used to perform the DSE, in the current implementation it parses the MULTICUBE[5] syntax.

The Application-Specific RunTime Manager is the main component of the framework that selects the best configuration (Plan element), within the list of OPs, according to a multi-objective constrained optimization that might involve observed metrics (using Goals) or design time computed metrics (using Static Goals).

Since the dynamic knobs are heavily application-dependent, is the application itself that is in charge to apply the configuration selected by ARGO(Execute element), closing the MAPE loop. In this way it is possible to deploy the framework in a wide range of applications, while minimizing the integration effort. In fact, we model the application as a sequence of different blocks of code that perform the elaboration iteratively. The idea is that at the beginning of each iteration, the application retrieves the configuration to use in the current iteration.

3 Framework integration

To employ separation of concerns, our workflow is based on three kind of files. The source code of the application describes the functional behavior, while we use two configuration files written in XML to express the adaptation layer: one file describes the design time knowledge and the third one describes the monitor infrastructure and the multi-objective optimization. ARGOuses a tool that automatically generates the glue-code required to integrate the framework in the target application.

To better clarify the required effort, Figure 2 provides an integration example considering a toy application. It shows the original source code written in black, while the integration code required to adopt ARGO is written in bold red. The application itself is very simple: on lines 9-16 the elaboration block, named "foo", performs the loop over the available jobs, while the function *do_job* (line 14) actually performs the computation.

```
#include ‘‘argo.hpp’’

int param;

int main ()
{
  argo::init();

  while( work_to_do() )
  {
    argo::block_foo
    {
      // do the computation
      do_job(param);
    }
  }
}
```

Fig. 2. This example shows how to integrate ARGO in an existing application code that exposes the elaboration as a loop, using the glue-code automatically generated by the framework tool from an XML configuration file.

In this example, we suppose that the elaboration is influenced by the parameter *param*, expressing the amount of processed data and representing the software knobs of the application. Since the code of toy application expose directly the elaboration loop, the integration requires only to include the created header file, initialize the framework and then wrap the execution call with the generated macro, highlighted in bold red. In this way the framework is able to observe and tune the elaboration block. Since no assumptions are made on the structure of the application code, the tool generates a hierarchy of methods to interact with the application, that requires to write the glue-code using more fine grained functions. In the worst case, the application designer is able to directly use the framework API.

4 Conclusion

In this work we have described a framework that enhances an application with an adaptation layer. In particular it adapts the knowledge base obtained at design time with the information gathered by the monitor infrastructure. Using this information, ARGO selects the best configuration according to the actual requirement of the application.

References

1. D. Gadioli, G. Palermo, and C. Silvano. Application autotuning to support runtime adaptivity in multicore architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 173–180. IEEE, 2015.
2. H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.
3. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
4. G. Palermo, C. Silvano, and V. Zaccaria. Respir: a response surface-based pareto iterative refinement for application-specific design space exploration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1816–1829, 2009.
5. C. Silvano, W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, S. Bocchio, R. Zafalon, P. Avasare, G. Vanmeerbeeck, et al. Multicube: Multi-objective design space exploration of multi-core architectures. In *VLSI 2010 Annual Symposium*, pages 47–63. Springer, 2011.
6. V. M. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore. Papi 5: Measuring power, energy, and the cloud. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 124–125. IEEE, 2013.