SELF-OPTIMIZING AND SELF-VERIFYING DESIGN: A VISION

Wayne Luk

Department of Computing, Imperial College London
180 Queen's Gate, London, UK
wl@doc.ic.ac.uk

ABSTRACT

This paper explores a vision of the design process in which components can optimize and verify themselves to improve efficiency, re-use, and confidence in correctness – the three design challenges identified by the International Technology Roadmap for Semiconductors. We illustrate what would take place for self-optimization and self-verification before and after deployment of the design, and present the benefits and challenges for the proposed approach.

1. INTRODUCTION

A good design is efficient and meets requirements. Optimization enhances efficiency, while verification demonstrates that requirements are met. Unfortunately, many existing designs are either inefficient, incorrect, or both.

Optimization and verification are recognised to be of major importance at all levels of abstraction in design. The 2005 International Technology Roadmap for Semiconductors listed "cost-driven design optimization" and "verification and testing" as two of the three overall challenges in design; the remaining challenge is "re-use".

What would a future be like in which these three challenges are met? Let us imagine that building blocks for use in design are endowed with the capability of optimizing and verifying themselves. A new design can be completed in the following way.

1. Characterize the desired attributes of the design that define the requirements, such as its function, accuracy, timing, power consumption, and preferred technology.
2. Develop or select an architecture which is likely to meet the requirements, and explore appropriate instantiations of its building blocks.
3. Decide whether existing building blocks meet requirements; if not, either start a new search, or develop new optimizable and verifiable building blocks, or adapt requirements to what can be realized.
4. After confirming that the optimized and verified design meets the requirements, organize the optimization and verification steps to enable the design to become self-optimizing and self-verifying.
5. Generalize the design and the corresponding self-optimization and self-verification capabilities to enhance its applicability and re-usability.

This article is dedicated to the memory of Professor Stamatis Vassiliadis, whose insights and leadership continue to be an inspiration. Many thanks to Kubilay Atasu, Tobias Becker, Ray Cheung, Andreas Fidjeland, Mike Flynn, Paul Kelly, Philip Leong, Henry Styles, Kong Susanto, David Thomas, and Steve Wilton for their help. The support of UK Engineering and Physical Sciences Research Council, European FP6 hArtes project, Celoxica, and Xilinx is gratefully acknowledged.

A key consideration is to be able to preserve self-optimization and self-verification in the design process: starting from components with such properties, the composite design is also self-optimizing and self-verifying. In the next few sections, we include more information about this approach.

2. OVERVIEW

Optimization can be used to transform an obvious but inefficient design into one that is efficient but no longer obvious. Verification can then show, for instance, that the optimization preserves functional behaviour subject to certain pre-conditions. A common error in design is to apply optimizations disregarding such pre-conditions. Verification can also be used to check whether a design possesses desirable properties, such as safety and security, to a particular standard.

Optimization and verification, when combined with a generic design style, supports re-use in three main ways. First, an optimized generic design provides abstraction from details, enabling designers to focus on the available optimization options and their effects. Second, a generic design offers choices at multiple levels of abstraction, from algorithms and architectures to technology-specific elements. Third, a verified optimization process improves confidence in the correctness of its optimized designs. Correctness must be established before a design can be re-used. In the event of errors, one can check whether the verification is incorrect, or whether the design is applied in a context outside the scope of the verification.

We take a broad view of self-optimization and self-verification. One way is to think of a design – which can include both hardware and software – and its characterization about the key properties that an implementation should possess. Such properties include functional correctness, type compatibility, absence of arithmetic underflow or overflow, and so on. The characterization can include prescriptions about how the design can be optimized or verified by specific tools locally or remotely. Various mechanisms, from script-driven facilities to machine learning procedures, can be used in the self-optimization and self-verification processes, making use of context information where available. Designers can focus on optimizing and verifying particular aspects; for instance, one may wish to obtain the smallest design for computing AES encryption on 128-bit data streams with a 512-bit key at 500MHz.

The proposed design flow involves self-optimization and self-verification before and after deployment (Table 1). Before deployment, compilation produces an initial implementation and its characterization. The characterization contains information about how the design has been optimized and verified, and also about opportunities for further optimization and verification; such opportunities can then be explored after deployment at run time for a particular context to improve efficiency and confidence of correctness.

Table 1. Context for pre-deployment and post-deployment.

	Pre-deployment (Section 3)	Post-deployment (Section 4)
focus	designer productivity	design efficiency
context	design tool environment, often static	operation environment, often dynamic
acquire context	from parameters affecting tool performance	from data input e.g. sensors
optimize/verify	optimize/verify initial post-deployment design	optimize according to situation
planning	plan post-deployment optimise/verify	plan to meet post-deployment goals
external control	frequent	infrequent

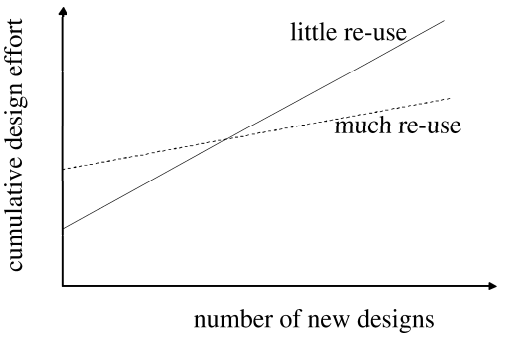

Fig. 1. Design effort: the impact of re-use.

The self-optimization of a design depends on context. Before deployment, the context is the design tool environment; the context can be acquired by identifying parameters that affect design tool performance. While automated facilities, possibly self-improving, attempt to figure out what combinations of libraries and tools would produce a design that best meets the requirements, designers can optionally control the tools to ensure such self-optimization and self-verification proceed in the right direction. In contrast, after deployment such external control is usually less frequent, for instance if the design is part of a spacecraft. To summarize, pre-deployment tasks are mainly strategic, and try to proactively determine possible courses of action that might take place at run time; post-deployment tasks are mainly tactical, and must choose between the set of possible actions to react to the changing run-time context.

Our approach has three main benefits. First, it enhances confidence in design correctness and reliability by automating the verification process. Second, it improves design efficiency by automating the optimization process and exploiting run-time adaptivity. Third, it raises productivity by enabling re-use of designs and their optimization and verification.

However, adopting systematic design re-use – especially when self-optimization and self-verification are involved – can require more initial effort than doing a one-off design. The designer needs to organize, generalize and document the designs appropriately. Only after some time, design re-use would become worthwhile (Figure 1). Moreover, there can be large overheads involved in supporting optimization and verification after deployment. In the long term, however, those who invest in capabilities for design re-use and design adaptability are likely to achieve substantial improvement in design efficiency and productivity.

3. PRE-DEPLOYMENT

Before deployment, a designer has the characterization of a desired design, and has access to building blocks and their characterization. The task is to develop an architecture that defines how selected building blocks are instantiated and composed to produce an initial design that either meets the requirements, or can be further optimized to do so, after deployment at run time. Post-deployment optimization and verification have to be planned carefully to avoid becoming an unaffordable overhead.

We assume that, at compile time before deployment,

1. the available computing resources are adequate to support the design and the tools, but
2. there is a limit on how much optimization and verification can take place since, for instance, some data values useful for optimization are only known at run time, and it is impractical to compute all possibilities for such values.

As a simple example, given that one of the two operands of an n-bit adder is a constant whose value is only known after deployment at run time, we wish to optimize the adder by constant propagation. It is,

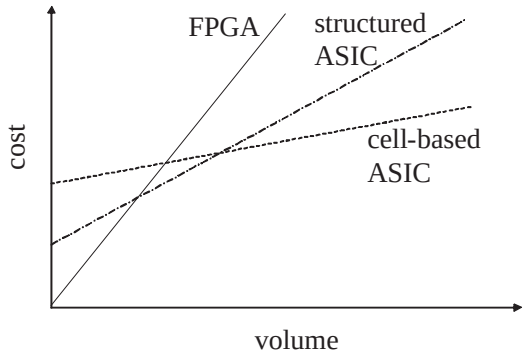

Fig. 2. Comparing cost and volume for FPGA and ASIC technologies.

however, impractical to pre-compute the configuration of all 2^n possibilities, unless n is a small number. Fortunately, if we target a bit-slice architecture, then it may suffice to pre-compute only two configurations for each of the n bits so that, at run time when the value is known, the appropriate configuration can be placed at the right location at the right time [26].

Designers may have to prioritize or to change their requirements until a feasible implementation is found. For instance, one may want the most power-efficient design that meets a particular timing constraint, or the smallest design that satisfies a given numerical accuracy. Other factors, such as safety or security issues, may also need to be taken into account.

Given that pre-deployment optimization is to produce an optimized design that would, where appropriate, be further optimized after deployment, the following are some examples of optimizations that can take place before deployment.

1. Choose a circuit technology in which the design would be implemented. The two common technologies are Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA); the choice of technology depends on volume and flexibility (Figure 2). For instance, cell-based ASIC tends to be cheaper at large volume since they have large non-recurring engineering cost, while FPGA is the other way round with structured ASIC somewhere in between. While ASIC technology can be used to implement adaptive instruction processors with, for instance, custom instruction extensions [4] or a reconfigurable cache [12], all the options for reconfiguration have to be known before fabrication. Adaptive instruction processors can also be implemented in FPGA technology [13],[35], which allows them much more flexibility at the expense of speed and area overheads in supporting reconfigurability.
2. Choose the granularity and synchronization regime for the configurable units. Current commercial FPGAs are mainly fine-grained devices with one or more global clocks, but other architectures are emerging: there are coarse-grained devices containing an array of multi-bit ALUs (Arithmetic Logic Units) executing in parallel [2],[14], as well as architectures based on self-synchronizing technology to enhance scalability [8]. Generally fine-grained devices have a better chance to be tailored to match closely with what is required. For instance, if a 9-bit ALU is needed, 9 bit-level cells in an FPGA would be configured to form that 9-bit ALU. For a coarse-grained device containing cells with 8-bit ALUs, two such cells would be needed. However, fine-grained devices tend to have a large overhead in speed, area, power consumption and so on, since there are more resources that can be configured. Coarse-grained devices, in contrast, have lower overheads at the expense of flexibility.
3. For instruction processors with support for custom instructions [4],[13], choose the granularity of custom instructions to achieve the right balance between speed and area. Coarse-grained custom

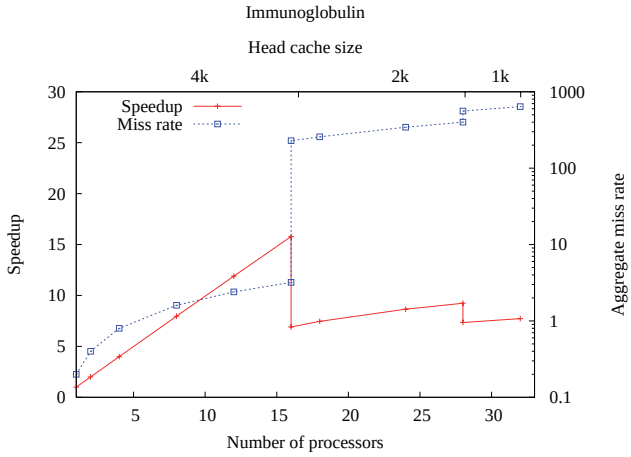

Fig. 3. Variation of speedup and aggregate miss rate against the number of processors for the Arvand multiprocessor system targeting the XC2V6000 FPGA.

instructions are usually faster but require more area than fine-grained ones. For instance, if the same result can be achieved using: (a) one coarse-grained custom instruction, or (b) 50 fine-grained custom instructions, then (a) is likely to be more efficient since there are fewer instruction fetch/decode, and there are more opportunities to customize the instruction to do exactly what is needed. However, since the more coarse-grained an instruction, the more specific it can become, there would be fewer ways for re-using a coarse-grained custom instruction than a fine-grained one.

4. Choose the amount of parallelism and hardware/software partitioning to match performance or size constraints by determining, for instance, the number of processing elements, the level of pipelining, or the extent of task-sharing for each processing element. Various factors, such as the speed and size of control logic and on-chip memory, and interfaces to other elements such as memory or sensors, would also need to be taken into account. As an example, Figure 3 shows how speedup varies with the number of processors targeting an FPGA for a multiprocessor architecture specialized for accelerating inductive logic programming applications [15]. Since the amount of FPGA on-chip memory is fixed, increasing the number of processors reduces the amount of cache memory for each processor; hence the linear speedup until there are 16 processors. After this optimal point, adding more processors reduces the speedup since the cache for each processor becomes too small.
5. Choose data representations and the corresponding operations. Trade-offs in adopting different kinds of arithmetic representations are well known: for instance redundant arithmetic tends to produce faster designs since no carry-chain is required, at the expense of size. Since fine-grained FPGAs support designs with any word-length, various static and dynamic word-length optimization algorithms can be used for providing designs with the best trade-off between performance, size, power consumption, and accuracy in terms of, for instance, signal-to-noise ratio [10]. Models and facilities to support exceptions, such as arithmetic overflow and underflow, should also be considered [21].
6. Choose placement strategies for processing and memory elements on the physical device, such as those interacting frequently are placed close to one another to improve performance, area and power consumption. It may be possible to automate the optimization of placement by a combination of heuristics and search-based autotuners [3] that generate and evaluate various implementation options; such methods would need to take into account various architectural constraints, such as the presence of embedded computational or memory elements [5].

Each example above has aspects that would benefit from verification, from high-level compilation [6] to flattening procedures [22] and placement strategies [23]. There are verification platforms [29] enabling consistent application of verification facilities such as symbolic simulators, model checkers and theorem provers. Such platforms show promise in supporting self-verification for complex designs, but much remains to be done to verify designs involving various technologies and across multiple levels of abstraction. Also, many of these platforms and facilities may be able to benefit from automatic tuning [17].

One important pre-deployment task is to plan self-optimization and self-verification after deployment. This plan would depend on how much run-time information after deployment is available. For instance, if some inputs to a design are constant, then such constants can be propagated through the design by boolean optimization and retiming. Such techniques can be extended to cover placement strategies for producing parametric descriptions of compact layout [22]. Another possibility is to select appropriate architectural templates to facilitate run-time resource integration [25].

Before deployment, if verification already covers optimizations and all other post-deployment operations, then there is no need for further verification. However, if certain optimizations and verifications are found useful but cannot be supported by the particular design, it may be possible for such optimizations and verifications to take place remotely, so that the optimized and verified design would be downloaded securely into the running system at an appropriate time, minimizing interruption of service.

4. POST-DEPLOYMENT

The purpose of optimization is to tailor a design to best meet its requirements. Increasingly, however, such requirements no longer stay the same after the design is commissioned; for instance, new standards may need to be met, or errors may need to be fixed. Hence there is a growing need for upgradable designs that support post-deployment optimization. Besides upgradability, post-deployment optimization also enables resource sharing, error removal and adaptation to run-time conditions – for instance selecting appropriate error-correcting codes depending on the noise variation.

Clearly any programmable device would be capable of post-deployment optimization. As we described earlier, fine-grained devices have greater opportunities of adapting themselves than coarse-grained devices, at the expense of larger overheads.

In the following we focus on two themes in post-deployment optimization: situation-specific optimization and autonomous optimization control. In both cases, any un-trusted post-deployment optimizations should be verified by light-weight verifiers; possible techniques include proof-carrying code checkers [34]. Such checkers support parameters that capture the safety conditions for particular operations. A set of proof rules are used to establish acceptable ways of constructing the proofs for the safety conditions.

As mentioned in the preceding section, should heavy-duty optimizations and verifications become desirable, it may be possible for such tasks to be carried out by separate trusted agents remotely and downloaded into the operational device in a secure way, possibly based on digital signatures which can verify senders' identity. Otherwise it would be prudent to include a time-out facility to prevent non-termination of self-optimization and self-verification routines that do not produce results before completion.

Besides having a time-out facility, post-deployment verification should be capable of dealing with other forms of exceptions, such as verification failure or occurrence of arithmetic errors. There should be error recovery procedures, together with techniques that decide whether to avoid or to correct similar errors in the future. For some applications, on-chip debug facilities [16] would be useful; such facilities can themselves be adapted to match the operational and buffering requirements of different applications.

Situation-specific optimization. One way to take advantage of post-deployment optimization in a changing operational environment is to continuously adapt to the changing situation, such as temperature, noise, process variation, and so on. For instance, it has been shown [31] that dynamic reconfiguration

that situation, although there could be power surges when the device is being reconfigured. Techniques have been proposed for FPGAs that would automatically adjust their run-time clock speed [7], or exploit dynamic voltage scaling [9]; related methods have been reported for microprocessors [11]. Such techniques would be able to take advantage of run-time conditions after deployment, as well as adapting to effects of process variation in deep-submicron technology.

A useful method for supporting situation-specific optimization is to integrate domain-specific customisations into a high-performance virtual machine, to which both static and dynamic information from post-deployment instrumentation is made available. Such information can be used in various situations for self-optimisation and self-verification, such as optimizing the way hardware or software libraries are used based on special properties of the library code and context from post-deployment operation.

Autonomous optimization control. "Autonomic computing" [19] has been proposed for systems that support self-management, self-optimization, and even self-healing and self-protection. It is motivated by the increasing complexity of computer systems which require significant efforts to install, configure, tune and maintain. In contrast, we focus on the design process that can support and benefit from self-optimizing and self-verifying components.

An evolving control strategy for self-optimization can be based on event-driven just-in-time reconfiguration methods for producing software code and hardware configuration information according to run-time conditions, while hiding configuration latency. One direction is to develop the theory and practice for adaptive components involving both hardware and software elements, based on component metadata description [18]. Such descriptions characterize available optimizations, and provide a model of performance together with a composition metaprogram that uses component metadata to find and configure the optimum implementation for a given context. This work can be combined with current customizable hardware compilation techniques [32], which make use of metadata descriptions in a contract-based approach, as well as research on adaptive software component technology.

Another direction is to investigate high-level descriptions of desirable autonomous behaviour, and how such descriptions can be used to produce a reactive plan. A reactive plan adapts to a changing environment by assigning an action towards a goal for every state from which the goal can be reached [30]. Dynamic reconfiguration can be driven by a plan specifying the properties a configuration should support.

Other promising directions for autonomous optimization control include those based on machine learning [1], inductive logic programming [15], and self-organizing feature maps [24]. Examples of practical self-adaptive systems, such as those targeting space missions [20], should also be studied to explore their potential for widening applicability and for inspiring theoretical development. It would be interesting to find an appropriate notion of verifiability for these optimization methods.

5. ROADMAP AND CHALLENGES

In the short term, we need to understand how to compose self-optimizing and self-verifying components, such that the resulting composite design is still self-optimizing and self-verifying. A key step is to provide both theoretical and practical connections between relevant design models and representations, as well as their corresponding optimization and verification procedures, to ensure consistency between semantic models and compatibility between interfaces of different tools.

It seems a good idea to begin by studying self-optimizing and self-verifying design in specific application domains. Experience gained from such studies would enable the discovery of fundamental principles and theories concerning the scope and capability of self-optimizing and self-verifying design that transcend the particularities of individual applications.

Another direction is to explore a platform-based approach for developing self-optimizing and self-verifying systems. Promising work [29] has been reported in combining various tools for verifying complex

7. REFERENCES

[1] F. Agakov et al, "Using machine learning to focus iterative optimization", *Proc. Int. Symp. on Code Generation and Optimization*, 2006, pp. 295–305.

[2] J.M. Arnold, "Software configurable processors", *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors*, 2006, pp. 45–49.

[3] K. Asanovic et al, *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183, 2006.

[4] K. Atasu et al, " Optimizing instruction-set extensible processors under data bandwidth constraints", *Proc. Design, Automation and Test in Europe Conf.*, 2007, pp. 1–6.

[5] T. Becker, W. Luk and P.Y.K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration", *Proc. IEEE Int. Symp. on Field-Prog. Custom Computing Machines*, 2007, pp. 35–44.

[6] J. Bicarregui, C.A.R. Hoare and J.C.P. Woodcock, "The verified software repository: a step towards the verifying compiler", *Formal Aspects of Computing*, 18(2), 2006, pp. 143–151.

[7] J.A. Bower et al, "Dynamic clock-frequencies for FPGAs", *Microprocessors and Microsys.*, 30(6), 2006, pp. 388–397.

[8] M. Butts, A.M. Jones and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing", *Proc. IEEE Int. Symp. on Field-Prog. Custom Computing Machines*, 2007, pp. 55–64.

[9] C.T. Chow, L.S.M. Tsui, P.H.W. Leong, W. Luk and S. Wilton. "Dynamic voltage scaling for commercial FPGA", *Proc. IEEE Int. Conf. on Field Programmable Technology*, 2005, pp. 173–180.

[10] G.A. Constantinides, "Word-length optimization for differentiable nonlinear systems", *ACM Trans. on Design Automation of Electr. Syst.*, 11(1), 2006, pp. 26–43.

[11] S. Das et al, "A self-tuning DVS processor using delay-error detection and correction", *IEEE Journal of Solid-State Circuits*, 2006, pp. 792–804.

[12] A.S. Dhodapkarm and J.E. Smith, "Tuning adaptive microarchitectures", *Int. Journal of Embedded Systems*, 2(1/2), 2006, pp. 39–50.

[13] R. Dimond, O. Mencer and W. Luk, "Application-specific customisation of multi-threaded soft processors", *IEE Proc. – Computers and Digital Techniques*, 153(3), 2006, pp. 173–180.

[14] C. Ebeling et al, "Implementing an OFDM receiver on the RaPiD reconfigurable architecture", *IEEE Trans. on Computers*, 53(11), 2004, pp. 1436–1448.

[15] A.K. Fidjeland and W. Luk, "Customising application-specific multiprocessor systems: a case study", *Proc. IEEE Int. Conf. on Application-Specific Sys., Architectures and Processors*, 2005, pp. 239–244.

[16] A.B.T. Hopkins and K.D. McDonald-Maier, "A generic on-chip debugger for wireless sensor networks", *Proc. First NASA/ESA Conf. on Adaptive Hardware and Systems*, 2006, pp. 338–342.

[17] F. Hutter et al, "Boosting verification by automatic tuning of decision procedures", to appear in *Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, 2007.