Methods for Evaluating and Covering the Design Space during Early Design Development

Matthias Gries

CAD-Group, Electronics Research Laboratory
University of California at Berkeley, CA 94720

Abstract

This paper gives an overview of methods used for Design Space Exploration (DSE) at the system- and micro-architecture levels. The DSE problem is considered to be two orthogonal issues: (I) How could a single design point be evaluated, (II) how could the design space be covered during the exploration process? The latter question arises since an exhaustive exploration of the design space by evaluating every possible design point is usually prohibitive due to the sheer size of the design space. We therefore reveal trade-offs linked to the choice of appropriate evaluation and coverage methods. The designer has to balance the following issues: the accuracy of the evaluation, the time it takes to evaluate one design point (including the implementation of the evaluation model), the precision/granularity of the design space coverage, and last but not least the possibilities for automating the exploration process. We also list common representations of the design space and compare current system and micro-architecture level design frameworks. This review thus eases the choice of a decent exploration policy by providing a comprehensive survey and classification of recent related work. It is focused on System-on-a-Chip designs, particularly those used for network processors. These systems are heterogeneous in nature using multiple computation, communication, memory, and peripheral resources.

Key words: Design space exploration, design space pruning, system-level design, micro-architecture design, design frameworks, benchmarking, multi-objective search
1991 MSC: 68-02, 68U07, 68Q35, 68U20

Email address: gries@computer.org (Matthias Gries).

Contents

1 Motivation

Having a look at today's common practice to design an integrated circuit or a whole system we recognize the impact of the designer's experience gained in prior design projects on the final system architecture. Indeed, taking the application domain of network processing as a prominent example, we see quite a diversity of available architectures in order to implement the same kind of application [1]. This variety of designs can rather be explained by the knowledge gained in recently completed, prior designs in each of the design teams than by application-driven architecture decisions. That means, given a specification of the application and system requirements, the design team shrinks the range of feasible designs to a small number of possible designs by

falling back on earlier, beneficial design decisions which might be sub-optimal for the current design problem and biased towards the designers' favored design style. The relative quality of the final design as a result of this ad-hoc system design approach compared to an optimal design will become even worse in the future due to the following trends:

- *Increasing complexity of the design:* The complexity of integrated circuits continues to follow Moore's law, thus doubling every 18 months. This in particular motivates to reuse prior design knowledge at higher levels of abstraction in order to cope with the sheer size of the design.
- *Heterogeneous architectures:* We see more and more heterogeneous architectures combining application-specific with general-purpose computing, different kinds of peripherals, and memory hierarchies. In addition, designers increasingly tend to use existing designs in parallel in order to fill the available area rather than to develop new and larger designs. Recognizing and exploiting the concurrency of applications therefore becomes a significant part of the design process. Besides this architectural diversity, more and more different technologies are being integrated onto a single chip, such as on-chip memory, analog interfaces, and high-frequency parts. It is therefore increasingly unlikely that a design team will be able to come up with an optimal solution by hand. Although a single designer could find an optimal subdesign of the overall system for his/her area of expertise (such as memories), naïvely putting optimal parts together does not necessarily lead to an optimal heterogeneous system.
- *Deep submicron effects:* A couple of effects which have been neglected during the design process in the past make the design quality worse, such as increasing interconnect delays and decreasing signal integrity. Again, this point underpins the growing dependency among different aspects of the design.
- *Decreasing time to market:* Last but not least the design window for success becomes smaller and smaller, thus increasing the pressure to reuse prior designs rather than developing new, optimized designs.

As a result of these tendencies we in addition discover that the development of software is often decoupled from the development of the hardware part if programmable parts are employed. A heterogeneous architecture as a result of ad-hoc integration of optimal subdesigns might therefore turn out to be virtually impossible to program and configure. The performance of the software on the final hardware might thus not meet the requirements on the final hardware as expected.

We see the following approaches to partly release the designer from the constraints imposed by the mentioned trends:

- Programmable, application-specific architecture building blocks are increas-

ingly used to replace ASICs, thus allowing their reuse for different application domains without sacrificing too much efficiency.
- Correct-by-construction methods, such as the automatic generation of a compiler from an architecture description language description, are employed to reduce the time needed for verification whether a design meets the specification.
- The development of software is introduced in an earlier phase of the overall design process. For instance, system-level design frameworks allow the modeling of the behavior of the software even at very abstract levels and retargetable compilers ease the evaluation of the actual software on the intended hardware at low levels of abstraction.

Although these techniques are able to relief the designer from complexity and time to market concerns to some extend, they are not sufficient to address increasing heterogeneity and a growing number of dependencies between subdesigns due to, for instance, deep submicron effects. As complexity increases, it is becoming more and more unlikely that an optimal design represents an 'intuitive' solution to the design challenge and it is therefore questionable whether an experienced designer could come up with a decent solution following the current ad-hoc design approach of pruning the design space by applying prior, favored design decisions. Consequently, a disciplined approach to design space exploration is needed in order to be able to evaluate large design spaces with a high number of potential designs. This includes algorithms to prune and cover the design space in a systematic way on different levels of abstraction and refinement. The goal of this paper therefore is to give a comprehensive survey and classification of recent work in the area of design space exploration of integrated circuits and systems to ease the choice of a problem-specific exploration policy. We compare the characteristics of different exploration methods and existing design tools. We also reveal areas of further research in order to improve the quantitative comparability of exploration techniques.

The paper is structured as follows. In the next section, we give an introduction to the problem of design space exploration. Section 3 continues with a discussion of methods for evaluating a single design point. In Section 4, a survey of approaches for traversing and covering the design space is given. In addition, techniques for design space pruning and automated exploration are reviewed. We give an overview of design representations used for DSE in Section 5 and summarize the properties of available frameworks in Section 6. Section 7 continues with a qualitative discussion of trade-offs involved by choosing appropriate evaluation and exploration methods. In Section 8 we suggest further steps to enable a more quantitative comparison of design space exploration algorithms. Section 9 concludes this paper.

2 Introduction

The term "design space exploration" has its origins in the context of logic synthesis. Clearly, a circuit can be made faster by spending more parallel gates for a given problem description (providing that the description offers enough parallelism) at the expense of area overhead. By extensively playing around with synthesis constraints, designer have been able to generate a delay-area trade-off curve in the design space defined by speed and area costs. This process of systematically altering design parameters has been recognized as an exploration of the design space.

Scheme for automated design space exploration: Y-Chart. Design space exploration tasks today often deal with high-level synthesis problems, such as the automation of resource allocation, binding of computation and communication to resources, and scheduling of operations, for varying design constraints, given a fixed problem description. In order to support early design decisions and due to increasing design complexity, exploration tasks are more and more performed on the system level. A systematic exploration often follows the Y-chart approach [2], see Fig. 1, where one or several descriptions of the application (including workload, computation and communication tasks) and one architecture specification are kept separately. An explicit mapping step binds application tasks to architecture building blocks. The following evaluation of the mapping, e.g., in terms of performance, area, or power consumption may require synthesis steps of the architecture description, rewriting/adapting application code, and dedicated compilation phases of the application onto the architecture in order to evaluate the design and perform (possibly simulated) test runs. Constraints from the architecture, application, and workload descriptions may influence the evaluation. Results from the evaluation may trigger further iterations of the mapping by adapting the description of the application and workload, the specification and allocation (meaning the selection of architecture building blocks) of the architecture, or the mapping strategy itself.

In this paper we will in particular focus on two kinds of methods:

- Methods that deal with the evaluation of a single design, represented by the performance analysis step in the Y-chart.
- Methods for the coverage of the design space by (more or less) systematically modifying the mapping and the allocation of resources, corresponding to the feedback paths from the analysis to the mapping and architecture representations in the Y-chart.

Methods for covering the design space only alter the description of the ap-

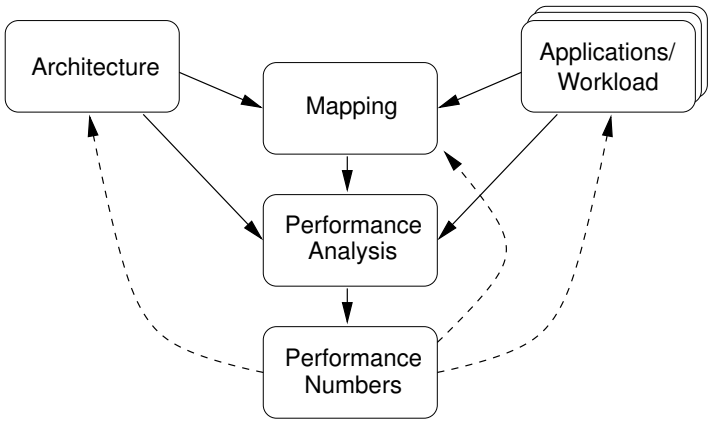

Fig. 1. Y-Chart approach to design space exploration.

plication to adapt or refine the description according to the facilities of the allocated architecture building blocks in order to ease a feasible mapping. During a design space exploration run, the functionality of the application usually remains unchanged and only the workload imposed by the application may vary.

Evaluating designs. Methods for evaluating a single design range from purely analytical methods, which can be processed symbolically, to cycle-accurate and RTL-level simulations which need complex executable models of the design under evaluation. Before a design can be evaluated, compilation and synthesis steps may be required, e.g. the hardware part of the design may be synthesized on an FPGA-based prototype. The complexity of validation phases can be reduced by correct-by-construction synthesis steps that guarantee correct implementations of the specification, thus avoiding validation by, for instance, simulation of test stimuli. In this case, the system evaluation using stimuli can focus on extracting design characteristics only, such as resource utilization.

Complexity of the exploration process. The solution space for a system-level design space exploration will quickly become large if arbitrary allocations and mappings are allowed. As a simple example, assume that b distinct hardware building blocks have been allocated and communication between these blocks is not a bottleneck. The application description may consist of t computation tasks. The building blocks are of a general-purpose type, such as CPUs with different micro-architectures. As a consequence, each task could potentially be mapped onto every hardware resource, leading to b^t feasible mapping choices. Thus, an exhaustive evaluation of all possible mappings quickly becomes intractable. Consequently, there is the need for automated and disciplined approaches to reveal a representative characterization of the design space without searching the design space extensively.

The complexity increases even further if multiple objectives are subject to the search. In order to evaluate one design whether it is Pareto-optimal (see Def. 4 in section 4.1) with respect to a set of solutions, all objective values of the design must exhaustively be compared with the corresponding objective values of every other design in the set. Fortunately, multi-objective explorations are usually bound to two or three objectives only, such as speed, costs, and power dissipation. The highest number of objectives that we found in related work is six [3].

It should be noted that exploration methods can work either on the problem space or the solution/objective space of the design. A system based on pre-designed IP-blocks can be optimized by, e.g., searching all possible combinations of parameters exported to the designer in the problem space, such as cache sizes and the clock frequency. Those parameters are part of the initial problem specification. In contrast to that, high-level synthesis methods are driven by constraints in the solution space, such as overall latency, power dissipation, and chip area.

Definition 1 (Problem space) *The problem space is defined by properties of the system that do not represent immediate design objectives but rather natural characteristics of the design space. In the context of DSE, the dimensions of the problem space often coincide with the axes of the architecture design space and may additionally include properties of the workload.*

A memory architecture, for instance, could be described by the required number of cache levels, the sizes of each level, and the caching algorithm used. An application could be represented by a task graph and an event model that triggers the tasks. All these specifications are part of the problem description and do not give any insights into primary objectives, such as the speed of the system. An exploration algorithm working on the problem space thus systematically chooses a system configuration, evaluates it, and decides whether this configuration is feasible or not. Finally, an optimal solution is selected in terms of one or more primary objectives, such as speed.

Definition 2 (Solution/objective space) *The solution space is defined by the primary objectives of the design space exploration, such as system cost, speed, and power dissipation.*

An exploration algorithm working on the solution space systematically constrains feasible designs in terms of primary objectives (see Section 4.2), i.e., the algorithm has to determine whether a suitable design can be found that fulfills the constraints while optimizing other objectives. Design parameters in the problem space are chosen accordingly, e.g. by logic synthesis algorithms.

In the following sections, methods for evaluating and exploring the design space are reviewed in more detail.

3 Methods for evaluating a single design point

In this section methods used to evaluate a single design point are discussed. Related work shows a variety of different approaches from detailed cycle-accurate and RTL-level simulations to purely analytical methods on relatively high abstraction levels. Depending on the chosen evaluation method, the mapping step in the Y-chart [2] (see Fig. 1) to determine performance values can be fairly complex involving explicit compiling or synthesis phases, whereas other methods represent mapping decisions implicitly by varying parameter sets. The orthogonal problem how to traverse the design space, given performance results for individual design points, will be discussed in the subsequent section.

Simulation-based evaluation can only estimate a single stimulus setting at a time, representing one particular implementation of a problem specification. The simulated workload must be chosen by the designer in a way that it represents a variety of typical working scenarios to avoid the optimization of the design for a special case. Analytical methods can help here since they are able to evaluate a design for a class of workloads (representing a range of stimuli for simulation-based tools) in a single pass. One drawback of analytical approaches however is that they often provide less precise results than simulation. Both evaluation techniques require a defined set of 'experimental' setups in order to produce performance-indicative, representative, reproducible, and comparable results during an exploration run. This procedure is often called benchmarking and is elaborated in the next subsection. Simulation-based and analytical methods for evaluating a design point are summarized in the following subsections.

3.1 Established benchmarks

In order to determine meaningful and comparable performance values, all evaluation methods need defined benchmarks of some kind to describe the workload imposed on the design under evaluation. In order to determine reproducible results, a benchmark includes a description of the application, a description of the architecture under test, constraints on the workload (e.g. defined by the working environment of the (embedded) system), a feasible mapping of the application onto the architecture, and defined metrics and cost functions. Available benchmarks can be classified according to their application domain. Examples are:

- *Network processing:* Related work on defining benchmarks for network processors include CommBench [4], NetBench [5], and activities of the Network Processor Forum (NPF, *http://www.npforum.org*). Related work on disci-

plined approaches for evaluating network processors can be found in [6,7]. The Internet Engineering Task Force (IETF) has a work group on benchmarking methodologies (BMWG) of internetworking technologies.

- *General purpose computing:* Benchmarks for general-purpose and scientific computing are published by the Standard Performance Evaluation Corp. (SPEC, *http://www.spec.org*). The Business Applications Performance Corporation (BAPCo, *http://www.bapco.com*) focuses on benchmarks for personal computers and notebooks.
- *Embedded systems:* Benchmarks for embedded systems including automotive, telecommunication, consumer, and control systems can be found in MiBench [8] and are also defined by the Embedded Microprocessor Benchmark Consortium (EEMBC, *http://www.eembc.org*).
- *Multimedia-centric computing:* Benchmarks focusing on multi-media processing can be found in MediaBench [9] and in the DSP-centric BDTI benchmarks from Berkeley Design Technology, Inc.
- *Database and transaction processing:* Business-oriented transactional server benchmarks are defined by the Transaction Processing Performance Council (TPC, *http://www.tpc.org*). SPECjAppServer2002 is a client/server benchmark from SPEC for measuring the performance of Java enterprise application servers in end-to-end web applications.
- *Parallel Computing:* Examples of benchmarks for parallel computing and multi-processing are SPEC OMP (OpenMP Benchmark Suite), Stanford Parallel Applications for Shared Memory (SPLASH2 [10]), and PARallel Kernels and BENCHmarks (Parkbench, *http://www.netlib.org/parkbench*).

3.2 Simulation-based evaluation

Simulation means to execute a model of the system under evaluation with a defined set of stimuli. A simulation can therefore only trace certain execution paths in the state space of the system that (hopefully) represent typical working modes of the design. Simulations are particularly well suited to investigate dynamic and sporadic, unforeseeable effects in the system, whereas formally verifiable systems require a deterministic behavior, given any stimuli. Results from analytical models can be too pessimistic since these models often consider the worst-case only. Simulations may reveal more realistic results for average-case optimization. One drawback of simulations is the need for an executable model. In an early phase of the design providing such a model may impose an unsubstantiated burden for evaluating early design decisions.

System-level simulation. The Ptolemy framework [11] can be used to model and simulate the interaction of concurrent system components by using different models of computation (MoC). Through hierarchical composition and

refinement the designer is able to specify software and hardware behavior at various levels of abstraction using different MoCs which are most suitable for the corresponding application domains.

Lieverse et al. [12] present an architecture exploration method based on a single MoC, namely Kahn process networks. The functional behavior of an application is kept separately from models describing the timing behavior of the architecture (e.g. CPU cores and buses). Applications are annotated with the computational requirement of one event. On the appearance of a computation event, the corresponding computation demand can be recorded in an execution trace. In the actual implementation of the tool traces are not recorded, but event demands are directly passed on to architecture models that associate a defined latency with each of these events. The actual assignment of computation events to architectural models determines a mapping of application processes to architecture components. The Artemis work described in [13,14] refines the work described in [12] in order to resolve deadlocks in Kahn process networks by introducing the concept of virtual processors and bounded buffers. One drawback of restricting the designer to using Kahn process networks is the inability to model time-dependent behavior. Moreover, since all scheduling decisions are implicitly taken following the organization of FIFO buffers, all resources are assumed to use FCFS schedulers on the event granularity level.

Thus, a natural alteration is used in [15] to model time-dependent workloads of network processors. Here, process networks with a notion of time are employed not only to model backlog in packet queues due to limited capacity of resources and the burstiness of packet arrivals, but also to implement time-dependent schedulers, such as Weighted Fair Queueing (WFQ). One further benefit of a notion of time is the option to investigate trading off the response time of a network processor versus the processing capabilities (i.e. resource load) of its computing resources by varying the length of loss-free event queues in front of schedulers.

Computation demands used as annotations for events could be determined by estimation, pseudo-code analysis, or even by isolated, fine-grained simulations of individual tasks to increase the accuracy at the system-level as, for instance, described in [16] and [17].

Cycle-accurate simulation. In order to increase the accuracy of evaluating a design, an often used level of refinement is defined by the precision of a single clock cycle. Cycle-accurate evaluation means that the timing is accurately modeled on a clock cycle basis. It does not necessarily imply an accurate replication of the behavior of the system. In order to emphasize accurate modeling of timing and behavior, we find the category of cycle-accurate, bit-true

simulation in literature. It means that at any given clock cycle, the state of the simulator is identical with the state of an actual implementation. Hardware models at this level of abstraction can either be based on software, modeling the timing and behavior of the hardware, or actual hardware descriptions in a hardware description language, enabling rapid prototyping on, for instance, an FPGA. The corresponding application under evaluation often is the actual application itself, i.e., it is described in a high level programming language or assembler, and not by a functional or behavioral model of the software. A cycle-accurate evaluation thus comprises either a co-simulation of the application together with the hardware description on a hardware simulator or an integrated simulation of the application on a software model of the hardware.

The Open SystemC [18] Initiative (OSCI, *http://www.systemc.org*) tries to leverage design knowledge in the C and C++ programming languages for system level modeling and evaluation. The goal is to provide an executable specification of hardware, software, and communication parts of a design early in the design process and to establish a path of refinements towards implementation, thus bridging the gap in current design practice between high-level models and hardware description languages. SystemC allows several layers of refinement and introduces notions of components (called modules), communication channels, hierarchical composition, processes, events and signals, as well as hardware-oriented data types – for instance bit level and logic types – in order to express concurrency, structural descriptions, communication, and synchronization. Future versions of SystemC will also incorporate primitives to express real-time operating system functionality and schedulers. SystemC's simulation library supports cycle-accurate evaluation. SpecC [19] also supports the iterative refinement of a design and is based on the C language. SpecC is an extension of C with additional hardware and software modeling constructs, whereas SystemC is a C++ class library. A detailed comparison of the capabilities of SystemC and SpecC can be found in [20]. Together with the richness of the C and C++ languages however also comes the potential drawback of versatility of implementations. That means, for instance, that SystemC models are not necessarily synthesizable. Support for SystemC has been integrated into a variety of tool flows, such as Synopsys' CoCentric System Studio, Axys Design's MaxSim Developer Suite, CoWare's N2C, and CoWare/Cadence's Signal Processing Worksystem (SPW). SystemC is increasingly being used to enable heterogeneous multiprocessor simulation by encapsulating proprietary simulators with modules and using bus wrappers for coupling the modules. SystemC-based tool flows are thus particularly well-suited for exploring interconnect structures and technologies, see [21–24].

Micro-architecture features of programmable processors are often investigated using cycle-accurate software models. Simulators, such as SimpleScalar [25] and SimOS [26], are specialized in certain classes of CPUs, such as MIPS-based cores. Variations of the micro-architecture that only affect run-time mecha-

nisms, such as caches, branch predictors, and issue widths, can be explored without the need of recompiling the application. Such a simulator therefore remodels a fixed IP-core, where only a small set of design parameters is exported to the user, such as the cache size. Modifications that affect the instruction set or variations of the data path however require a retargetable compiler that needs to compile every application to all potential micro-architectures. These mapping and modification steps may be eased and automated using architecture description languages (ADLs). One of these ADL-based design environments is being developed in the Mescal project [27,28]. Another example of ADL-based architecture exploration is described in the work of Mishra et al. [29–31] based on the EXPRESSION language [32]. Further examples of ADLs are LISA [33], nML [34], MIMOLA [35], and Facile [36]. ADLs can be distinguished according to the family of architectures they are able to express (e.g. single vs. multi-threaded), the ability to integrate effects of the micro-architecture (e.g. pipelining), and their options to support automated design space exploration by, for instance, an explicit mapping step and support for retargetable code, simulators, and synthesizable hardware generation. ADL-based frameworks often allow the automatic generation of cycle-accurate software models of the hardware and also partly support the automatic creation of synthesizable hardware descriptions. Surveys of ADLs can be found in [37,38].

The automatic generation of efficient micro-architecture simulators has recently been covered by several papers. ADL-based simulator generation is described in, e.g., [39,40] for LISA, in [41] for Expression, and in [36] for Facile. Techniques vary from interpretative simulation, interpretative simulation with caching of previous evaluations, to compiled simulation, where for each benchmark a separate simulator must be generated. Compiled simulation is the fastest but also most inflexible technique, which requires a large memory footprint, since all instructions of the benchmark are predecoded and part of the simulator. Interpretative simulators, which decode instructions during run-time of the simulation, are most flexible since one simulator can be reused for multiple benchmarks. The drawback is of course slow simulation speed compared to compiled simulation. A simulation technique in between these two extremes is interpretative simulation with caching, sometimes also called just-in-time compiled simulation. Here, the simulator keeps a cache of the results/effects of recently evaluated expressions in order to accelerate the execution of following appearances of these expressions. In the Facile-based tool flow [36], the so-called action cache works on the granularity of a simulation step. Entries of the cache are indexed by run-time static input, such as pipeline state and the instructions being simulated. In the LISA case [40], cache entries are indexed by program address and represent whole instructions. Cache-based simulation speed of course depends on the size and organization of the cache, trading off memory footprint and performance.

Examples of retargetable compilers – a prerequisite for automatic, cycle-accurate design space exploration of processor micro-architectures – targeted at embedded systems [42] and ASIPs are CoWare's LISATek (*http://www.coware.com*) [43], Chess/Checkers (*http://www.retarget.com*) [44], FlexWare [45], and Tensilica's XCC (*http://www.tensilica.com*). These compilers in particular specialize on code density, power issues, and reliability.

3.3 Combination of simulation-based and analytical methods

In order to reduce the overhead involved with the simulation of a complete system under evaluation, the following methods try to reduce simulation time by gathering all characteristics, which are common between designs being evaluated and which are not subject to the design space exploration, into a single initial simulation. Information extracted from the initial simulation run can be reused by all evaluations. The evaluation time narrows to the time it takes to evaluate distinctive features.

Trace-based performance analysis. This kind of performance estimation is in particular common for evaluating cache and memory structures, see the survey in [46]. An initial program run extracts all memory accesses and stores them in a trace. Given a cache model, the trace can then be used to calculate hit and miss statistics as well as overall performance estimates. The aim of this procedure is to save evaluation time by only doing an expensive simulation once. Different cache structures can be evaluated by reusing the same trace data collected from the initial simulation. Exemplary studies and tools that use this method for performance and energy analysis driving the design space exploration of a memory subsystem are by Fornaciari et al. [47,48] and Givargis et al. [49].

In the work presented by Lahiri et al. [50–52] this technique is applied to the design of on-chip communication structures. An initial system-level simulation of communicating components representing the workload environment, i.e. HW and SW components surrounding the communication structure under investigation, is performed to collect traces of communications going on between components. Traces are compressed by generating a communication analysis graph (CAG) that accumulates burst transfers and computations into single events. Given a communication substructure template consisting of point-to-point and shared connections as well as bridges, the CAG is modified to incorporate effects of synchronization, arbitration, and block transfers. Modifications include adding nodes and adjusting time stamps of nodes, which enables the estimation of performance and resource utilization.

Zivkovic et al. [53] augment traditional traces, that usually contain information on data transfers and task executions only, with control information in order to evaluate the cost of control as well.

Analytical models with initial, calibrating simulation. The analytical approach described by Franklin and Wolf [54] requires an initial characterization of benchmarks using exhaustive simulation runs for a range of cache organizations. Extracted information from these runs like miss rates and load and store instruction shares are fed into analytical models which allow reasoning about resource utilization, area requirements, and performance. The approach has been extended in [55] to include power requirements.

3.4 Purely analytical approaches

Analytical methods come into play if deterministic or worst-case behavior is a reasonable assumption for the system under evaluation. In addition, building an executable model of the system as well as simulations might be too costly or even impossible at the time of the evaluation. Analytical models thus in particular ease early design decisions by identifying corner cases of potential designs.

Static profiling. Well established methods for static program analysis, such as the complexity analysis of algorithms, the dependency analysis of a static schedule of a task or function call graph to extract worst-case behavior, or simply counting of operations appearing in pseudo code, can be used for performance estimations of an application mapped onto an architecture, as for instance performed in [56]. The reader is referred to standard literature in Computer Science, such as books by Knuth [57] and Sedgewick [58], to learn more about complexity analysis and common data structures for established sorting and searching algorithms. Analytical approaches that take certain elements of the micro-architecture of a programmable processing core into account can be found in the domain of worst-case execution time (WCET) estimation for embedded systems. The application models usually require the absence of sporadic and non-deterministic effects, such as the presence of interrupts, recursion, and operating systems. The analysis works on assembler, pseudo-code, or a high-level programming language description and therefore often has to neglect the impact of certain compiler optimizations. The architecture description is bounded to a single processor with simple pipeline and cache models. The formulation as an integer linear program often constitutes the core of the analysis as, for instance, described in the papers by Li et al. [59] and Theiling et al. [60].

Event stream-based analytical models. For certain application domains dedicated calculi, task, and workload models exist which allow symbolic evaluation of a design. Richter et al. [61] (and the references therein) give an overview of mature analytical techniques for evaluating the task execution on shared resources for event streams, such as periodic events, periodic with jitter, and sporadic preemptions. They extend these techniques by coupling the analytical models using event model interfaces, thus enabling system-level evaluations of platform-based designs. If coupling existing analytical models is not a concern, calculi provide a generalized approach to real-time embedded system design, potentially providing tighter bounds to end-to-end delays and storage bounds of shared memory implementations. One example is the network calculus [62] which has been applied to network processor design [63–65] and used for evaluating real designs [66,67]. Its applicability has also been shown for real-time embedded systems in general [68]. Recent extensions include the distinction of event types and dependencies within and among event streams [69] and variable execution demands per event [70]. The more details are included into analytical models, the more the borderline between simulation and analysis is blurred, since additional characteristics on the level of discrete events might be required. That means, analytical evaluation can trade off the complexity of the evaluation with the accuracy and general applicability of the model. Although the analytical evaluation of a system might become more computational demanding than a simulation, the main advantage of a simplified specification remains, since an executable model is not required. Analytical models are thus well suited for early design decisions, where corner cases have to be identified quickly, maybe even automatically by using parameterized models.

High-level synthesis. The evaluation of an application-specific architecture given a task graph may require an explicit synthesis step in order to extract area requirements and detailed timing information. The classical high-level synthesis problems of allocating resources, binding computations to resources, and scheduling operations under timing and/or resource constraints are either solved by exact methods, such as integer/mixed linear program formulations [71,72], or by heuristics, such as ASAP and ALAP [73,74], list- [75–77], force-directed scheduling [78,79], or evolutionary algorithms [75,79] (mainly used for allocation and binding problems).

In conclusion, the trade-offs involved by choosing an appropriate evaluation method are shown in Fig. 2. Analytical models allow a fast evaluation of a relatively large fraction of the design space, thus enabling the identification of corner cases of the design. Over several possible steps of refinement with increasing effort for evaluation and implementation the design space can be bound to one particular design point. This funnel representation resembles

the upper part of the platform-based design double pyramid [80], i.e., the final design point could also represent a whole platform. Methods for systematically exploring the design space on one of the layers of abstraction are discussed in the following section.

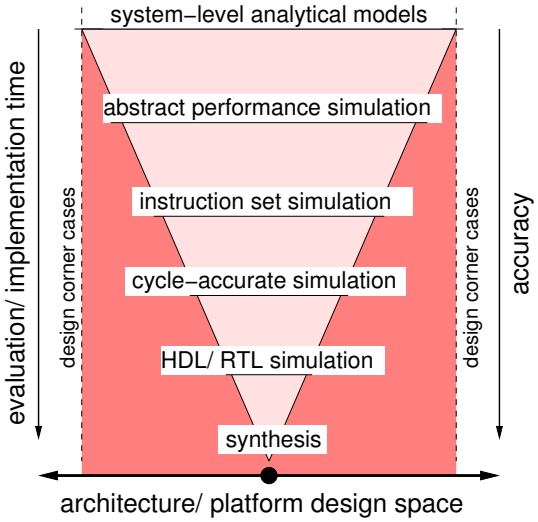

Fig. 2. Design funnel model. Refinements of the evaluation method narrow the reachable design space (vertical direction), whereas covering algorithms explore the size of the design space (horizontal direction).

4 Methods for exploring the design space

After having discussed methods used to evaluate a single design point, this section provides a survey of algorithms used to walk through and reasonably cover the design space. Exploring the design space is an iterative process which is usually based on the Y-chart [2] approach. Here, application and architecture descriptions are explicitly associated to each other in a mapping step and evaluated afterwards. The mapping could include compile and synthesis phases to enable the performance analysis. Results from the evaluation of that particular design point could then be used to further guide the exploration by varying application and architecture descriptions as well as the mapping between the two.

In the following sections we provide a coarse classification of search strategies depending on the number of objectives that are active during the exploration, a review of common cost functions and metrics, a survey of recent work on search strategies and design space pruning techniques, and a list of supporting functions for automated exploration in the area of computer architecture design.

For the classification of methods we need the term of Pareto optimality [81] which is introduced next. This property only has a meaning if a multi-objective search of the design space is performed. In the area of micro-architecture design, objectives could be the minimization of costs, power consumption, or the maximization of the speed. These objectives may show tight connections between each other. Thus, optimizing with a single objective in mind may reveal severe trade-offs with respect to the other objectives.

Definition 3 (Pareto criterion for dominance) *Given k objectives to be minimized without loss of generality and two solutions (designs) A and B with values $(a_0, a_1, \ldots, a_{k-1})$ and $(b_0, b_1, \ldots, b_{k-1})$ for all objectives, respectively, solution A* dominates *solution B if and only if*

$$\forall_{0 \leq i < k} i : a_i \leq b_i \quad \textit{and} \quad \exists j : a_j < b_j \ .$$

That means, a superior solution is at least better in one objective while being at least the same in all other objectives. A more rigorous definition of *strict dominance* requires A to be better in all objectives compared to B, whereas the less strong definition of *weak dominance* only requires the condition $\forall_{0 \leq i < k} i : a_i \leq b_i$.

Definition 4 (Pareto-optimal solution) *A solution is called* Pareto-optimal *if it is not dominated by any other solution. Non-dominated solutions form a* Pareto-optimal set *in which neither of the solutions is dominated by any other solution in the set.*

That means, designs in the Pareto-optimal set cannot be ordered using Def. 3. Thus, all elements in the set define reasonable solutions and they are subject to further decision constraints in order to choose a design for a given problem. An example is visualized in Fig. 3. The two-dimensional design space is defined by cost and execution time of a design, both to be minimized. Six designs are marked together with the region of the design space that they dominate. Designs 1, 4, 5, and 6 are Pareto-optimal designs, whereas design 2 is dominated by design 4 and design 3 by all other designs, respectively. Without further insights into the design problem all designs in the set {1,4,5,6} represent reasonable solutions.

Optimization methods can now be classified according to the following criteria (see [82] and the references therein):

- *Decision making before search:* The designer decides how to aggregate different objectives into a single objective (cost) function before the actual search

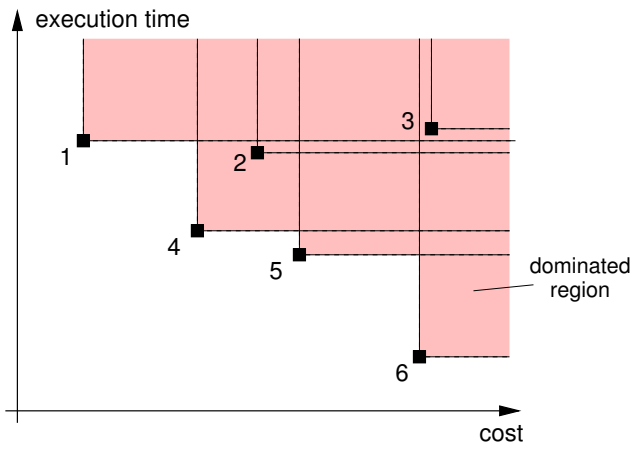

Fig. 3. Two-dimensional design space with Pareto-optimal designs 1, 4, 5, and 6.

is performed. In this way, well-established optimization methods can be applied. However, e.g. by using a weighted sum of objectives, certain regions of the design space may no longer be reachable by the search method (see the example below). Another procedure would be to convert certain objectives into constraints for an optimization problem with a reduced number of objectives. A non-arbitrary aggregation of objectives requires some knowledge about the design space to find a solution which is not sub-optimal. This might be inconsistent with one of the major goals of DSE, i.e. the determination of characteristics of the design space.

- *Search before decision making:* The search for optimal solutions is performed with multiple objectives in mind which are kept separate during the search. The result of the search is a set of Pareto-optimal solutions. Only after the search additional criteria or preferences are applied to find an optimal solution for a given problem. In this way an unbiased search can be done and problem-specific decisions only require the set of solutions. Hence, a single search may serve several problem-specific decisions (no rerun of the search required).
- *Decision making during search:* This category is a mixture of the two preceding groups. Here, initial search steps may be used to further constrain the design space and/or guide the search to certain regions of the design space. These steps may be repeated iteratively. Constraints and/ or guidance can be derived automatically or interactively by presenting intermediate search results to the designer.

Thus, the choice of a single- or a multi-objective search algorithm not only influences the point of time when design objectives are defined, but also affects the whole exploration process. Using a single-objective search, the result of the optimization is a single design point. That means, searches must be repeated with, for instance, varying weights or constraints on the objective function in order to explore the design space and generate a set of Pareto-optimal solutions. Depending on the shape of the objective function that aggregates

several objectives, certain regions of the design space might not be reachable at all. In the example in Fig. 4 the region of feasible solutions in the design space defined by cost and latency is shaded. Designs one, two, and three represent possible Pareto-optimal solutions. Assume that a weighted sum of the objectives x and y is used as the objective function f, i.e.

$$f(x, y) = a \cdot x + b \cdot y \qquad a, b \in \mathbb{R}_0^+ \tag{1}$$

with some weights a and b. The overall goal is to minimize f, i.e.

$$c \stackrel{!}{=} f(x, y)|_{min}$$

Equation 1 can be transformed to

$$y = \frac{c}{b} - \frac{a}{b} \cdot x =: c' - a' \cdot x$$

and describes a straight line in $x - y$ space. As we can see in Fig. 4, the ratio of the weights a and b therefore defines the constant slope of the line, whereas the optimization goal to minimize f moves the line towards the origin. Two different optimization runs are shown where we were able to find the solutions one and three. It is clear from the picture that we will never be able to reach solution two with any combination of the weights a and b since the straight line will always hit the solutions one or three in order to minimize f (represented by c). In a similar fashion one can also graphically show that the reduction of objectives by converting objectives into constraints in fact reduces the reachable space of solutions.

Contrary to that, multi-objective searches are potentially able to find all three Pareto-optimal solutions in a single optimization run. The actual choice for one of the solutions depends on further constraints or objective functions that apply combinations of the objectives used for the search. A typical application scenario could be the exploration of an IP-core library in terms of, for instance, cost and performance for a certain application domain. Suitable solutions depending on the actual application could be the fastest solution, the cheapest solution or the solution with the best cost/performance ratio. All three solutions can be derived from the set of Pareto-optimal solutions without rerunning the search.

4.2 Objective/ cost functions and metrics

In the following, widely-used objectives for design space exploration are pointed out. Single-objective optimizers tend to use a weighted sum, ratio, or product

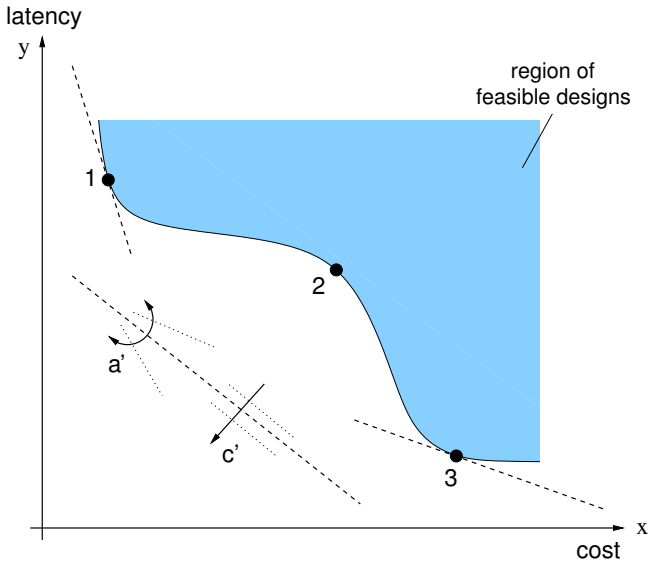

Fig. 4. Finding Pareto-optimal designs using a weighted sum of objectives.

of several objectives in order to consider conflicting criteria, whereas multi-objective algorithms can keep objectives separately so that the results of the search are not biased towards a certain region of the design space.

4.2.1 Primary objectives

Primary objectives concern the properties of the overall system and are typically used directly as optimization goal, i.e., they do not represent an intermediate, supportive cost metric, but drive the optimization process directly. The objectives listed in this category can universally be applied to design space exploration and are not domain-specific.

- *Cost:* The cost of a design could be measured as the sum of all component costs integrated in the system, e.g., based on the wholesale prices from different component manufacturers or based on the own manufacturing costs which could, for instance, be determined by the area consumption in the target technology and the packaging costs. Fixed costs, such as the fabrication costs (e.g. mask sets) or the engineering costs for designing the system, cannot drive the optimization process and are therefore often not included.
- *Power dissipation:* Optimization for power more and more becomes the focus for design space exploration. On the one hand, high-end systems optimized for speed have to cope with the generation of heat within the system that degrades the life-time of the components. On the other hand, embedded systems not only focus on the minimization of the worst-case power dissipation, but also on the power leakage during idle periods of the system in order to decrease the costs for maintenance, e.g., by extending the life-time of batteries.

- *Speed:* The speed of a design can be expressed by different metrics, such as the throughput achieved for computations and communications, the overall amount of data processed or transferred, the latency/ response time for certain events (whether deadlines are met), the period length of a schedule of computations and communications, or the bare clock speed supported by the design.
- *Flexibility:* The flexibility of a design can be seen as a meta-objective since it is difficult to express this metric quantitatively. However, many fundamental design decisions are based on the need of programmability or dynamic reconfigurability in order to extend the life-time of a design, to be able to incorporate late fixes due to, for instance, changes in communication standards, or to ease the remote maintenance of an embedded system. A first step towards defining quantitative metrics for flexibility can be found in the work by Haubelt et al. [83]. An application is described as a hierarchical DAG where hierarchy levels represent different options (algorithms) to implement the same functionality denoted by the parent node in the graph. An architecture is supposed to be more flexible the more options described in the application DAG it can implement, given timing and cost constraints. This definition of flexibility however requires that all possible functionality of the application can be enumerated in advance. Often, flexible architectures are however used in order to be able to implement functionality unknown at design time. In order to determine an upper bound on flexibility with this definition, the flexibility of a programmable architecture would only be bounded by the memory space restricting the number of possible programs.

4.2.2 Secondary objectives

Metrics in this category are either focused on the properties of only a part of the overall design or provide supportive information on the design, i.e., they reveal characteristics of the design that influence primary goals. The utilization of resources, for instance, could be seen as one component of the primary cost or power dissipation objectives. Secondary objectives are often problem-specific and facilitate the analysis of the overall design, pointing the designer to bottlenecks of the design. Metrics listed under primary objectives could, of course, also be applied to only parts of the design in order to support the analysis of the system. It should be noted that there are also optimizer-specific metrics that guide the search, such as the steepness to surrounding solutions in the case of hill climbing or the number of dominated solutions in the case of some multi-objective, evolutionary algorithms. The discussion of optimizer-specific metrics is beyond the scope of this paper. Common secondary objectives are:

- *Utilization of computation and communication resources:* The utilization of

a resource determines the fraction of the overall execution time of a benchmark during which the resource is busy with processing the benchmark. Depending on the primary goal and the application domain, the goal of the optimization could be to maximize the utilization in order to exploit the silicon area as much as possible. Reducing the utilization however could lead to more power-efficient solutions that could also provide headroom on flexible architectures for further extensions of the application.

- *Static/ dynamic profiling results:* Given an executable specification of the application, different kinds of profiling information could be extracted to guide further design decisions. An example is generating a histogram of the instructions used, e.g. data transfer vs. control vs. computation vs. bit level operations, that could indicate further exploration steps towards certain architectures supporting the most frequent operations in hardware. Profiling results could therefore be used as an affinity metrics towards certain design decisions.
- *Affinity metrics:* Affinity metrics defined by Sciuto et al. [84] determine whether an executable specification of the application favors DSP-, ASIC-, or general purpose-like computing solutions as architecture component. Exemplary metrics are the multiply-accumulate degree, I/O ratio, and bit manipulation rate of the reference application.
- *HW-SW partitioning specific metrics:* Metrics in the domain of hardware-software partitioning can be seen as a special case of affinity metrics with two design choices only. For a given problem description it must be decided which parts of the specification should be implemented in software and which in hardware. Indicators, such as potential speedups, area- and communication overheads, locality and regularity of computations as, for instance, defined in [85,76], are used for guiding the decision towards hardware or software.
- *I/O- and communication-specific metrics:* Apart from primary speed objectives, such as throughput, latency, and the number of transactions, I/O-specific metrics include the number of I/O stall cycles and arbitration penalties which affect the primary speed metrics.
- *Memory-specific metrics:* Cache characteristics that in particular represent the speed of the memory subsystem for a given application include the number of conflict and capacity misses, cache hit and miss ratios, as well as the locality of accesses extracted from the application. Metrics reflecting cost and power dissipation properties are the code size and the memory consumption of the application, e.g., generated from the maximum processing backlog of a task graph.
- *Reliability:* In the domain of embedded systems reliability may become as important as cost and power dissipation objectives since it might be virtually impossible to service the remote system or since hard real-time functionality must be preserved under all circumstances. Reliability comes at the price of over-provisioned and/or redundant designs.
- *Deterministic behavior:* In the domain of hard real-time systems, determin-

istic behavior might be a primary objective in order to fulfill safety requirements. Deterministic behavior is achieved by dimensioning the design for the worst-case behavior of the system. All dynamic and sporadic events must only have bounded effects on the design, i.e., the behavior must be predictable.
- *Physical size:* The physical size and weight of a design may be of primary importance for embedded systems in, for instance, the automotive domain and particularly affects the cost of the design.
- *Compatibility:* In order to partly retain the investment in previous designs, the compatibility of software, or the computing infrastructure to the new design becomes important. For the hardware part of the new design this could mean to maintain a constant interface to the surrounding computing environment, whereas compatible software requires a constant programming model for the user. That means, compatibility usually comes at the price of suboptimal cost and/ or power dissipation objectives.
- *Usability:* Usability describes the ability of a design to ease its initialization, configuration, and programming towards the deployment in a certain application domain. Usability could also include the properties of the user interface.
- *Testability:* The support for testing a circuit can be a considerable cost factor of the design. Hardware building blocks must be included or adapted to allow a test of the circuit using externally generated test vectors. Partial or full scans must be supported by test points and scanable registers and might also require the support of certain standardized test modes, such as JTAG boundary scan. A system can also provide circuitry to generate test vectors internally, which is called built-in self test.

4.2.3 Combined metrics

In particular single-objective optimizers combine several objectives in order to consider conflicting criteria. Multi-objective algorithms could essentially also use combined objectives in order to reduce the number of dimensions to the problem, i.e., it could make sense to only consider the speed-cost and the flexibility-cost ratios for a certain design and not speed, cost, and flexibility as separate optimization goals. The most prevalent combined objectives are:

- *Energy-delay product:* The energy-delay product is in particular used to assess embedded systems. The power requirements are trade off against the speed of the design with the overall objective to reduce the product.
- *Computations-power ratio:* This objective can be interpreted as a computational density related to power dissipation. Designs are supposed to be better than others if they achieve more computations for a given power budget or consume less power for a given speed. The ratio between the number of computations and the power dissipation for a defined benchmark

reflects this.

- *Speed-cost ratio:* This combined objective represents a computational density related to the costs of the design. A design is better than other designs if it achieves higher speed at the same price or the same speed at a lower price. The ratio between the speed of a design and its cost combines this behavior in a single objective.
- *Flexibility-related:* In the same way the performance of a design has been combined with cost and power objectives in the preceding combined objectives, ratios and products can be defined to express the trade-off between flexibility and cost, speed, or power. Examples are the flexibility-power ratio, the flexibility-cost ratio or the flexibility-delay product.

In the following, we review methods for exploring the design space that can be employed having a single or multiple primary objectives in mind.

4.3 Strategies for covering the design space

In this subsection, related work concerning the question how one could cover the design space is discussed. The mentioned categories are not strictly orthogonal to each other. In the following Subsection 4.4 methods for reducing the size of the design space are revealed that can be used in combination with any of the approaches presented in this subsection in order to decrease the exploration time.

Exhaustively evaluating every possible design point. This straightforward approach evaluates every possible combination of design parameters and therefore is prohibitive for large design spaces. The design space can be reduced by limiting the range of parameters and/or by parameter quantization. Multiple objectives can easily be maintained. The search process is completely unguided and unbiased towards preferences of the designer. Examples of design systems and case studies based on exhaustive search include system-level simulation [15,16,86], high-level synthesis [87,73,88,89,78,90,72], ADL-driven approaches [30,91], cycle-accurate simulations [92,93], instruction set simulators [49], code parallelization and partitioning onto multi-processors [94], trace-based analysis [50], and last but not least static analysis [56].

Randomly sampling the design space. Evaluating only random samples is the obvious choice for coping with large design spaces. It also has the advantage of revealing an unbiased view of the characteristics of the design space. In [95], a Monte Carlo-based system is described where random samples of the

solution space are generated by randomly creating constraints for logic synthesis. Another approach is to apply simulated annealing techniques. Starting from an initial design, changes to the design become more and more unlikely with advancing search time. In [96], Srinivasan et al. compare explorations driven by simulated annealing with results using an evolutionary approach. Gajski et al. [97] combine an exhaustive search over all possible architecture allocations with a simulated annealing-based exploration of mappings for each allocation.

One could also think of combining the mentioned random walks with principles from Tabu search [98] in order to avoid evaluating the same design twice. Tabu search thus enforces diversification into unexplored regions of the design space and also incorporates mechanisms to explore around interesting design points found so far (the so-called intensification phase of the search). Tabu search however would require additional, possibly computational intensive maintenance operations in order to keep track of recent moves and bad strategic choices. Tabu search has its roots in operations research and has only seldom been applied to electronics design so far. We are not aware of any related work using Tabu search in the context of electronics design. Moya et al. [99] compare Tabu search with simulated annealing for an artificial, two dimensional optimization scenario. Tabu search appears to be more robust against discontinuities and errors in the cost function and more effectively covers "bumpy" terrain than simulated annealing. Tabu search is also the more promising search strategy compared with simulated annealing in the architecture allocation problem investigated in [100].

Incorporating knowledge of the design space. Search strategies in this category try to improve the convergence behavior towards (Pareto-) optimal solutions by incorporating knowledge of characteristics of the design space into the search process. The knowledge may be updated with every iteration of the search process or may be an inherent characteristic of the search algorithm itself. All mentioned methods are heuristics.

Hill climbing, for instance, evaluates the neighborhood of the current design to determine the steepest next step towards the optimization goal. In order to avoid being trapped on top of a local maximum, hill climbing requires backtracking mechanisms which might be expensive in "bumpy terrains". Moreover, the search becomes aimless on plains and is not able to recognize diagonal ridges since the probe directions would always lead to lower quality solutions. In [51], hill climbing is used to explore the mapping of communication onto channels. In [101], a kind of hill climbing is one of the investigated techniques to explore VLIW micro-architectures.

Evolutionary search algorithms combine random walk with survival-of-the-

fittest ideas while constructing new generations of a set (population) of solutions. Better solutions are more likely to survive from generation to generation and new solutions can be either created by mutation (random walk) or crossover of existing solutions. Crossover tries to combine features from two good solutions to generate even better solutions. In the way the designer chooses a representation of the problem and implements the mutation and crossover operations (working on those representations) already guides the search. For instance, mutation and crossover operations may generate new solutions that are not feasible. A repair mechanism could then prefer certain features of the solution over others. Thus, domain knowledge may inherently guide the search. Naïve implementations of those operations may however also avoid certain regions of the design space to be reached. Moreover, problem-specific representations require recoding of the evolutionary operations for other problem domains. Single-objective evolutionary algorithms have been used in [96,77] and multi-objective evolutionary searches are described in [63,75,102–104,3,79,105]. Dick et al. [79] combine an evolutionary search with simulated annealing so that allocation and binding changes are less likely to happen with an increasing number of iterations.

Searches may also iteratively be guided by distance measures or other means of affinity towards certain regions of the design space. Sciuto et al. [84] define affinity metrics for applications towards mappings onto DSPs, ASICs, and general-purpose processors. Peixoto et al. [85] define metrics which favor resource sharing. Those metrics guide optimizations towards clusters of similar computations that show high locality. In this way, the communication between clusters is minimized, whereas resource sharing is maximized.

Path-oriented versus unguided search. This distinction emphasizes how the search progresses from iteration to iteration. Path-oriented searches are, for instance, hill climbing and evolutionary algorithms (implementing crossover). Exhaustive searches and random samples like unsupervised Monte Carlo methods belong to the class of unguided searches. Again, the latter class aims to give an unbiased view of the design space, whereas algorithms from the former class use domain knowledge of the design space to guide the search. Path-oriented walks may have the advantage of potentially reusing intermediate results of earlier design evaluations along the path. The underlying assumption here is that a design only slightly changes from one step to the next so that most of the evaluation experience from the previous design can be reused for the evaluation of the current design.

Single design at a time versus set-oriented search. This property differentiates between the number of designs that must be kept available in each iteration to perform the search. Exhaustive searches and random walks only

look at one solution at a time, whereas methods exploiting domain-specific knowledge tend to use several designs at a time to find an improved design. Hill climbing and evolutionary algorithms thus belong to the latter category.

In summary, Fig. 5 graphically describes different search strategies for covering the design space. A discrete design space defined by two design parameters (in problem space) or two design constraints (in solution space) P_1, P_2 is assumed. Otherwise, an exhaustive search would already constitute a subsampling of the design space in this figure, since the parameters would have been quantized before the search.

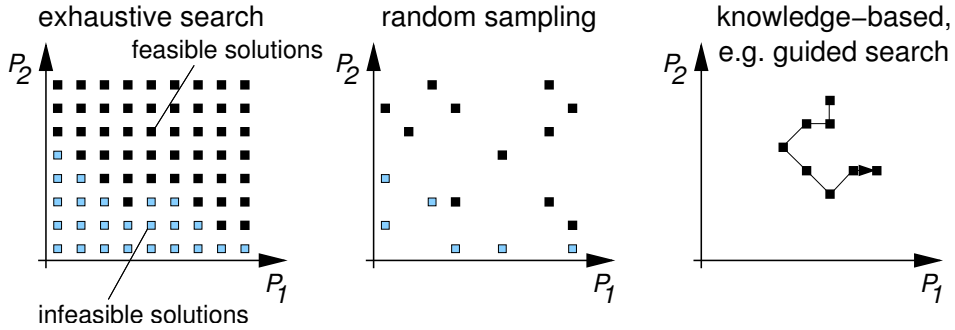

Fig. 5. Common approaches for covering the design space. A discrete, two-dimensional design space defined by two design parameters/ constraints P_1 and P_2 is shown.

4.4 Pruning the design space

All mentioned exploration methods can employ further techniques to reduce the complexity of the search by pruning the design space. Several practical approaches have been described in the literature, as will be described next.

Hierarchical exploration. Starting with a coarse problem statement interesting regions of the design space are identified and ranked. Refined models are used to explore those regions. The higher-level models could also be back-annotated with results from the refined explorations to improve the high-level characterization of the design space. The search thus switches back and forth between high- and low-level explorations.

Hekstra et al. [93] use a single simulation and profiling run of an exemplary VLIW architecture to extract timing information for all possible VLIW designs of their architecture library. Corner cases of the design are simulated to determine bounds on specific design parameters. This procedure is called *probing*. Results from probing and profiling are used to determine the design

parameters which influence the solution most. Only those parameters are explored exhaustively, whereas the remaining parameters are considered with a sensitivity analysis which will be introduced later.

Mohanty et al. [17] use an analytical step first to prune the design space by symbolic constraint satisfaction. This information is used to limit the design space for trace-driven system-level simulations. Cycle and power accurate simulators may be used on the lowest level of abstraction. Results from accurate single component simulations can also be back-annotated to elements of the system simulation to improve results on a higher layer of abstraction.

In [74], different solutions of logic synthesis are explored. The problem description, e.g. a flow graph, is subdivided into templates of apparent regular structures, i.e. clusters of operations which can be found again and again in the graph. Results from exploring those templates can then be applied to all instances of the corresponding template. A further step searches the design space at the granularity of the supergraph consisting only of templates.

Baghdadi et al. [16] use a few individual building blocks which are synthesized to RT level in order to extract timing information for possible mappings on a higher level of abstraction, i.e., this information is back-annotated to a higher level of abstraction.

Subsampling of the design space. Subsampling the design space is a reasonable choice if the designer is interested in an unbiased exploration where an exhaustive search would be prohibitive. The subsampling pattern could be completely random, based on some regular grid, or biased by some expected shape of the design space and/or objective function(s). Monte-Carlo based searches, simulated annealing, and evolutionary optimizers (implementing mutation) use random subsampling patterns. All approaches which quantize design parameters to reduce the design space, e.g. by allowing only a set of fixed bit widths for architecture building blocks, apply a regular sampling pattern. This property translates, for instance, to the length of one step using hill climbing. In [101], defined sweeps across the design space of VLIW architectures are used to explore the design space. Fig. 6 shows some common sampling patterns.

Subdividing the design space into independent parts for optimization. The goal of this approach is to reduce the number of possible designs by dividing the optimization problem into independent subproblems. In this way, we do not need to consider all possible combinations of design parameters but rather all combinations of Pareto solutions found for the subsystems

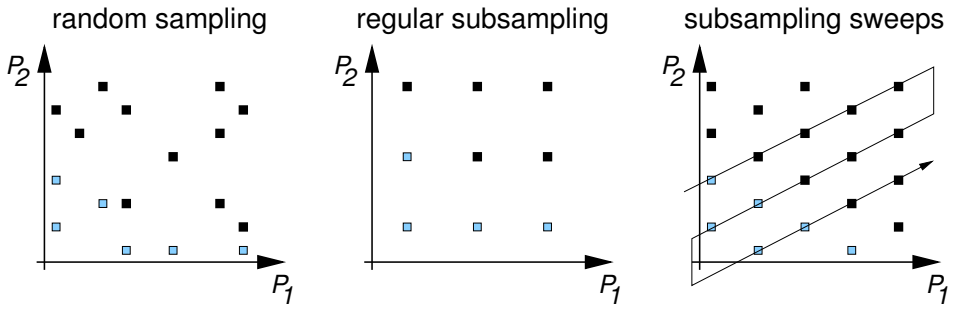

Fig. 6. Common approaches for subsampling the design space.

(Fig. 7). Although the dimensions of the solution spaces for the different subsystems could be similar, e.g. cost and speed, the problem space of each of the subsystems usually comprises quite different, domain-specific parameters. A cache could be described by its size and organization, a VLIW core by its issue width and operator bit width, etc.

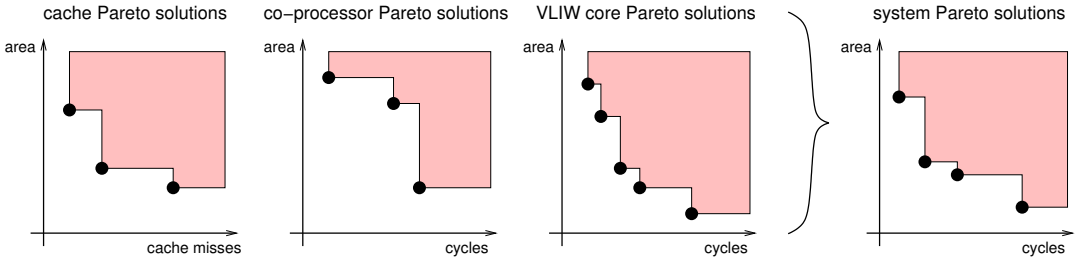

Fig. 7. Subdividing the design space into independent parts for optimization.

Kathail et al. [106] divide the optimization of an embedded computing system into separate optimizations of the cache memory hierarchy, a customized systolic array used as a co-processor, and a VLIW processor. The exploration of memory subsystems described in [107,108] separates optimizations concerning power consumption (by determining the cache size), main memory size (by varying the data layout), and speed (by optimizing address calculations). Givargis et al. [109] describe a method where the designer initially defines clusters of design parameters which affect and depend on each other. Separate clusters can be explored independently. Then, Pareto-optimal configurations from clusters are merged.

Sensitivity analysis of design parameters. The underlying assumption of this approach is the independence of design parameters. A sensitivity analysis of the design space is done using a set of reference benchmarks. In separate calibration runs only a single parameter is varied at a time, whereas all the other parameters are set to (fixed) arbitrary values. With each calibration step, the dynamic range of solution properties, such as power consumption and speed, is recorded.

Formally, given n design parameters P_i, $0 \leq i < n$ with C_i possible configurations each, the number of possible configurations C for the overall system is the product of all parameter configurations

$$C = \prod_{i=0}^{n-1} C_i.$$

In order to reduce the number of configurations for evaluation, we perform the following experiments for each parameter P_i for all r reference benchmarks b_j, $0 \leq j < r$:

- Set all other parameters P_k, $0 \leq k < n$, $k \neq i$ to arbitrary (fixed) values $P_k := P_{k_0}$.
- Evaluate the system for all possible configurations C_i of parameter P_i and note the results of interest, for instance, in terms of speed.
- The system's sensitivity S_i to parameter P_i is then defined as the difference between the maximum and the minimum result in the series of results for all configurations of parameter P_i, i.e. in this example the difference between the maximum and the minimum achievable speed by varying P_i.

Given these results for the set of reference benchmarks, the sensitivities could be averaged among all benchmarks for each parameter. The design parameters P_i can then be sorted in decreasing order of sensitivity $\{P_{S_{n-1}}, P_{S_{n-2}}, \ldots, P_{S_0}\}$, where $P_{S_{n-1}}$ denotes the parameter with highest sensitivity and P_{S_0} the parameter with lowest sensitivity, respectively. The complexity of the design space exploration using these parameters can now be reduced by evaluating only designs defined by the sum of all parameter variations (versus multiplying all parameter variations in the exhaustive case) and by dropping parameters with small influence on the solution space from the exploration. For example, assuming that we stop evaluating the design space at parameter P_{S_q}, the resulting number of evaluations C' to perform for an unknown setup is given by

$$C' = \sum_{i'=S_q}^{S_{n-1}} C_{i'}.$$

Work described in [103,104,48,93] uses sensitivity analysis to prune the design space. Ascia et al. [103] show one approach to extend sensitivity analysis to multiple objectives.

Constraining the design space. This straightforward task is listed as a separate point since the identification of design space constraints can form a significant initial step of a DSE run. The identification of corner cases of the

design space could be done, for instance, by probing the design space or by worst-case analytical methods, such as the network calculus [62,63] for the network processing domain and event-stream based methods for the real-time embedded domain [61].

4.5 Supporting functionality for automated DSE

The following approaches address practical issues that arise when dealing with implementing automated DSE, i.e. integrating legacy tools, working on several layers of abstraction, and reducing the resource requirements of the evaluation host computer.

Hierarchical simulator integration. The evaluation of IP-core based designs may require the integration of simulation frameworks from different manufacturers which may work on different levels of abstraction, e.g. functional versus cycle-accurate levels. In [110], a framework is presented that integrates different simulators. Problems that are specific to this integration are addressed, such as a description of how stimuli at low abstraction levels could be generated from high-level stimuli and how the state of low-level simulators can be maintained between simulation runs that are triggered by higher level simulators.

Validation: Equivalence check. A DSE run aimed to explore implementations for a tightly defined specification requires automated validations whether an implementation still meets the requirements of the specification. In the general case, formal verification tools address this problem. There also exist less complex approaches for well defined subproblems. For instance, in [111] a procedure is described to check two designs for equivalence of output traces. Searching the state space of both designs is avoided. Another partial verification technique is symbolic simulation (see [112] and the references therein) where the state space is subsampled by verifying system properties for defined symbolic inputs only. The effort spent for validation and verification can be reduced by employing correct-by-construction techniques. Architecture description languages, for instance, keep a central description of the micro-architecture of a processor to automatically derive correct compilers, simulators, and hardware descriptions from it.

Automatic refinement of the task graph. Given a DSE problem which should be solved using the Y-chart [2] approach, specifications for applications

may not necessarily match the features exported by the architecture description. Thus, in order to perform the mapping step, automatic adaptations of the specifications are required. In [113], automatic task graph refinements to cope, e.g., with synchronization and serialization of accesses are described. The same kind of problem is addressed in the work by Lieverse et al. [114] for trace-based simulation. In [50,52], automatic refinements of communication traces are described to consider DMA block lengths and bus protocols. This principle is sketched in Fig. 8a.

Trace compression. Trace-driven simulations may require a large number of long traces. This is why research efforts also focus on compression and abstraction techniques for traces. Lahiri et al. [50,52] show how communication traces could be abstracted from, e.g., burst transfers to only reveal abstract communication events as sketched in Fig. 8b.

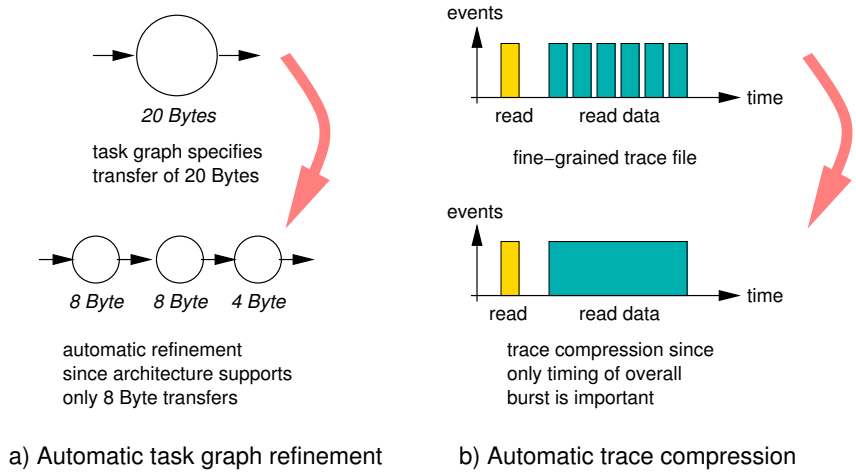

Fig. 8. Supporting functionality for automated DSE.

Synthetic trace generation. In order to relief the designer from managing and recording traces for trace-based simulation, the synthetic generation of traces according to profiling information is described in [115] in order to use (legacy) trace-based tools.

Coupling of incompatible building blocks. While designing a heterogeneous multi-processor system the designer faces the problem that he/she has to use architecture building blocks from different manufacturers and sources. As a consequence, the blocks might have been modeled at different levels of abstraction and provide different communication interfaces. So-called wrappers/ adapters [116,117] are needed for simulation and synthesis to fit components

together, which can affect hardware and software parts of the system. Knowledge from interface synthesis [118–121] can be reused for the hardware part of a wrapper, whereas following the approach of Networks-on-Chip [117] based on standard network interfaces affects protocol stacks on programmable cores. A theory of adapters for event-stream models can also be used for analyzing heterogeneous systems [122].

Operating system customization. If operating systems are used on programmable cores to relieve the application programmer from details of the hardware and provide services at run-time, the problem arises that traditional general-purpose operating systems offer too much functionality, do not meet hard real-time constraints, and have large resource requirements. Apart from porting an existing third party real-time operating system to the system under development and leveraging knowledge from systematic device driver generation [123,124], the generation/synthesis of a lean operating system, e.g. starting from a primary kernel of an open-source OS, could also be an option [125,126].

User-assisted DSE. This point is in conflict with the preceding points since it assumes that completely automated DSE cannot effectively be performed but requires regular interaction with the designer to guide the search to certain regions of the design space. This is why Hu et al. [127] deal with the projection and visualization of a multi-objective design space to a three dimensional representation to ease interactive design decisions by the designer.

5 Representing the design space

Numerous representations at different abstraction levels are used to model applications and architectures for enabling design space exploration following the Y-chart. For each category, the following subsections review the most prominent examples. A couple of these representations have already been introduced in preceding sections. This section mainly serves as a summary of available representations for the sake of completeness of this DSE survey.

5.1 Architecture models

The listed architecture models mostly focus on performance perspectives of the modeled hardware. The same kind of representations could also be used to

model or enable the analysis of properties like power, memory, and area consumption. Models can also be classified as abstract or executable descriptions. Abstract models only represent performance symbolically by, for instance, associating the required latency in clock cycles with each operation without actually executing any hardware description. Executable specifications on the other hand allow to more precisely model state-dependent behavior, such as the timing of caches and pipelines.

Abstract, instruction-accurate performance models. The timing behavior of a resource is described by a list of symbolic instructions and their associated latencies. Traces of symbolic instructions are generated by annotated application models during execution and handed to the architecture models to determine the overall execution time of an application. This kind of performance model is used in the SPADE [12,53] framework and in the case study presented in [15]. A special case of this kind of performance model is often used by logic synthesis-based methods. Here, functional units, that support a set of operations, are represented by their fixed execution delay and, for instance, by their area consumptions using a particular technology for synthesis. Blythe et al. [73] use this representation.

Abstract, task-accurate performance models. The timing behavior of a resource is described by a list of supported tasks and their worst-case or average (estimated) execution times on this resource. This abstraction level is often used to characterize System-on-a-Chip designs which are subject to system-level design space exploration where the granularity of interest is on the level of computation cores, memories, and buses. The abstraction could further be increased by associating execution times with complete platforms. Abstract performance models at the task level are, for instance, used in [79,75,72].

Abstract, non-linear, accumulative service descriptions. Service curves (see [62] and the references therein) describe a non-linear worst-case envelope for the computation or communication capabilities of system-level components for all possible time intervals. They are therefore able to express properties like stalls, batch processing, and arbitration policies. Service curves are measured in units of, for instance, cycles, instructions, bytes, or service time per second. Service curves are used in EXPO [128] to model building blocks of SoC designs. Service curves can be used together with arrival curves, that will be introduced in the next subsection, to determine system properties analytically, such as the utilization of resources and the response time to certain events under a defined workload.

Micro-architecture templates. On the one hand, micro-architecture templates constrain the design to a certain class of architectures, e.g. VLIW-style computation for general processing cores. On the other hand, templates allow a more specialized generation of compiler and simulator tools and thus potentially enable more optimized soft- and hardware code generation. The PICO framework [106] uses templates for caches, programmable computing cores, and coprocessors. Other examples in the area of micro-processor architecture simulators include SimpleScalar [25] and SimOS [26] which are focused on defined micro-architecture classes. Lahiri et al. [50] use templates for interconnect structures on SoCs. Preconfigured IP-blocks can also be seen as architecture templates, revealing only design parameters to the designer that lead to feasible designs.

Specification in a hardware description language (HDL). HDLs offer different levels of abstraction in order to describe functionality and behavior of an architecture, including the levels of abstraction mentioned earlier. SystemC, for instance, enables functional, transaction level, and cycle-accurate models, whereas other HDLs, such as Verilog and VHDL, are in particular used to describe the actual structure of the underlying hardware in the form of RTL netlists.

Specification in an architecture description language (ADL). ADLs allow modeling of computing architectures on higher levels of abstraction than HDLs, e.g. on the pipeline- or instruction-level, while preserving paths to cycle-accurate simulators and synthesizable hardware models. ADLs usually focus on a certain class of architectures that they are able to express in order to enable the efficient generation of software compilers for that architecture class. ADLs can be distinguished according to the family of architectures they are able to express (e.g. single vs. multi-threaded), the ability to consider effects of the micro-architecture structure (e.g. pipelining), and their options to support design space exploration by, for instance, providing retargetable code, simulator, and synthesizable hardware generation. Examples of ADLs are LISA [33], EXPRESSION [32], nML [34], MIMOLA [35], and the languages used within the Mescal framework [27,28] and for the Liberty simulator generator [129]. Surveys of ADLs can be found in [37,38].

5.2 Application models

Application models offer different levels of abstraction in the same way as architecture models do. The specification could again be abstract, i.e., the model

defines a workload without executing the specification. Executable specifications could use different abstraction levels: they could represent the actual application or , e.g., a discrete-event performance model of the application.

Kahn process networks (KPN, [130]). Concurrent processes communicate through FIFO-organized, unbounded, uni-directional point-to-point channels. Each process itself performs sequential computations on local data. Computations are interleaved with read and write requests from / to channels. Read requests are blocking, i.e., the process stalls until sufficient data is available on the channel, whereas write requests are non-blocking due to unlimited channels. Given an input stream of data the result after executing the KPN is deterministic, i.e., the result does not depend on the order of execution of processes. The YAPI model [131] extends KPNs by associating a data type with each channel and by introducing non-determinism by allowing dynamic decisions on selecting the next channel communication so that, for instance, scheduling on shared resources can be modeled. KPNs are used in SPADE [12,53,114] and Artemis [14]. In [15], process networks with a notion of time are used to describe packet processing workloads and to enable the analysis of time-dependent behavior, such as traffic managers and dynamic deadline-based schedulers. Finally, the Compaan framework [132] enables the automatic transformation from nested loop programs written in Matlab to Kahn-like process networks that are especially suitable for FPGA implementations.

Directed acyclic graphs (DAGs). DAGs are often used in the context of logic synthesis. Nodes represent atomic operations (or whole non-preemptive tasks) and directed edges data dependencies between operations, respectively. A DAG thus represents the data flow of an application. The work by Blythe et al. [73] uses DAGs on the granularity of logic operations, whereas in [96,76,72] tasks are used as nodes in the DAG. In the work by Blickle et al. [75] the nodes of a DAG represent computation and communication tasks. The EXPO framework [128] uses DAGs to represent packet processing applications for network traffic flows at the granularity of tasks.

DAGs with periods and deadlines. Directed acyclic graphs (DAGs) representing computation tasks are annotated with execution deadlines and periods for each DAG. A multi-rate system thus comprises several DAGs with different associated periods. This application description is used by Dick et al. [79].

Synchronous data flow (SDF) In an SDF graph, nodes represent atomic operations/ tasks and directed edges data dependencies between nodes. In addition to a DAG, each SDF node is annotated with produce and consume numbers that represent the number of data tokens produced and consumed by the computation of the node. This information is static, which is why feasible schedules for the execution order of the nodes and memory requirements for buffering tokens (if a bounded schedule exists) can be derived at compile time.

Control data flow graphs (CDFG) and dynamic data flow (DDF). CDFGs can be extracted from the source code description of a program, i.e., they reveal all options the control flow could take at run-time of the application. The data flow part of the graph shows the underlying concurrency of the application, whereas the control flow part determines synchronization points of the data flow and dynamic decision points at run-time. Control flow nodes are branch and loop constructs. CDFG representations are used in the comparison in [53]. A similar model with the same expressiveness as CDFGs is a dynamic data flow (DDF) graph. A DDF graph contains SDF nodes and nodes with data-dependent dynamic behavior. The additional node types are switch, selection, and repeat. Each of these nodes has a special input. The value of a token arriving at this input determines, to which of several outputs an incoming data token should be handed over (switch node), from which of several inputs a token should be handed over to the output (select node), and how many times an incoming token should be replicated on the output (repeat node), respectively. DDF is a generalization of boolean data flow (BDF). BDF only contains switch and select nodes as dynamic nodes that offer two choices only, represented by a boolean value at the special input. Integer-controlled DDFs are used by Artemis [14].

Non-linear, accumulative workload/ event stream descriptions. Arrival curves (see [62] and the references therein) describe a non-linear worst-case envelope for event streams, such as network traffic for all possible time intervals. They are therefore able to express properties like periods, bursts, sporadic events, and jitter. An example is shown in Fig. 9 d). Note that in the domain of real-time systems separate event models exist for the mentioned properties (see [61] and the references therein). Arrival curves can be used to model the workload imposed on an application or system as well as the output generated by the system (black box view). Arrival curves together with DAGs are used in the EXPO tool [128].

High-level programming language descriptions. Algorithms are usually specified in high level procedural or object-oriented programming languages, such as C, C++, and Java, or more application-specific languages,

such as Matlab for signal processing. It is therefore often essential to support the application specification in these languages in order to be able to use common benchmarks, although the capabilities of those languages to express concurrency are limited. Frameworks using high-level programming language descriptions as application description are, for instance, Milan [17], PICO [106], and ASIP-Meister [133].

Transaction Level Modeling (TLM). TLM is a discrete-event model of computation used by SystemC, where modules interchange events with time stamps. TLM is used to model the interaction between software and hardware modules and communication through shared buses. The modules themselves can be specified at different levels of abstraction. The application might be specified as a functional, un-timed state machine model, whereas the architecture might represent an instruction-accurate performance model. A transaction aggregates several 'traditional' events, that usually represent features of an implementation, and thus raises the level of abstraction in order to improve evaluation speed. As an example, an application model may issue a read access request as a transaction to a memory architecture model. Inside the memory model, this transaction represents a sequence of activation, read, and precharge events.

Communication analysis graphs (CAG). In the work by Lahiri et al. [50] a CAG is used as an intermediate representation which is input to performance analysis. The CAG represents a compressed version of communication and computation traces extracted from system-level simulations. It is therefore a DAG containing communication and computation nodes, which includes timing information. A CAG can thus also be seen as an abstract task level performance model of the application that includes a schedule.

Co-Design Finite State Machines (CFSM) [134]. The communication between CFSM components is asynchronous, whereas within a finite state machine (FSM) the execution is synchronous, based on events. The FSM component models sequential behavior, whereas the asynchronous communication between components allows the expression of concurrency. Communication takes place over buffers with a single element only that can potentially be overwritten. That means, the transmission of new data has precedence over the transfer of older data and the loss of old data can be tolerated. This application model is motivated by characteristics of the automotive domain, where the processing of recent sensor data has high priority and a component may ignore certain events.

Click model of computation [135]. Click is a representation especially suited for describing packet processing applications. Nodes represent computation on packets, whereas edges represent packet flow between computations. The packet flow is driven by push and pull semantics. Push transfers are initiated by traffic sources and pull transfers are driven by traffic sinks. Queue elements terminate both push and pull transfers. The StepNP framework [24] uses Click as application description and in Mescal [27] Click is one possible domain where applications can be specified.

Hierarchical and heterogeneous models of computation. The Ptolemy framework [11] allows to combine various models of computation hierarchically in order to model and evaluate concurrent components. Metropolis' [136] metamodel language also allows to express different kinds of models of computation, such as transaction-level modeling and process networks.

Fig. 9 shows a couple of common representations that are used to describe applications and their workload. Kahn processes communicate through unbounded FIFOs, whereas in DAGs – in this example a node represents a one-clock arithmetic function – a connector implies one register. In the Click model, buffers synchronize between push and pull paths. Representations a)-c) are used to describe the application and could be combined with arrival curves shown in d) to model the overall workload; e.g., an event stream bounded by the curves in d) could trigger push paths in Click or the inputs of a DAG.

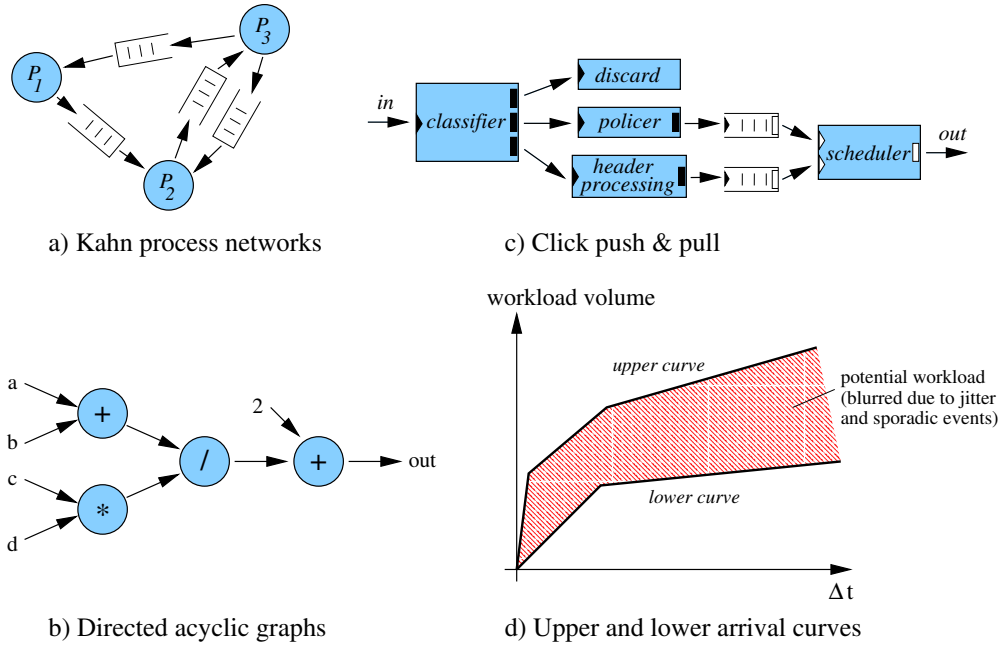

Fig. 9. Examples of application and workload representations.

Although methods for simplifying the deployment of designs is beyond the scope of this paper, we want to shortly mention the problem field of defining a suitable programming model for a configurable design, since the definition of a programming model may strongly depend on the chosen application and architecture representation.

A programming model should provide an abstraction from the underlying hardware, so that applications can be specified in a convenient way for the application domain of the design. A good programming model supports an intuitive abstraction of the architecture while providing the necessary expressiveness to achieve efficient implementations. That means, given an application described by the constructs of a programming model, an efficient implementation on the hardware can be derived by (possibly a series of) transformations from the model down to the representation of the programmable hardware.

In the domain of general-purpose computing, we are used to high-level programming languages that provide us, e.g., with constructs to express loops and object-oriented program flows. We rely on the fact that the underlying CPU has a program counter and hardware building blocks to ease and accelerate the processing, such as caches and pipelines. We, however, do not explicitly address these features on the abstraction level of the high-level programming language, unless it is required to implement the desired program flow by, e.g., implicitly modifying the program counter with a conditional branch construct. The transformation from a program down to the representation of the hardware (registers, functional units, instructions) is automated and performed by a compiler.

When we think about developing a programming model for domain-specific processors or whole SoC designs with multiple cores, the application models described in the preceding subsection might already represent a decent programming model for the underlying architecture. They could also be used as an intermediate representation, comparable to the intermediate representation of a front-end/back-end compiler. Finally, they could be augmented with additional constructs to explicitly represent hardware building blocks. An example of the latter case is NP-Click [137], where the Click application model has been extended to export hardware threads and the data layout to the programmer while hiding specific resource sharing mechanisms. In Intel's programming model [138] for network processors, the programmer specifies the functionality of the application with a kind of process network. The programmer annotates each process with worst-case critical path information and performance requirements. The data layout between several memories must be done explicitly. Examples for further application domains are Synchronous

Data Flow (SDF) models for signal processors [139] and computations organized in streams for media processors [140]. Gropp et al. [141] summarize common programming model abstractions for parallel computing, such as threads of control, message passing, and shared memory computations. Finally, in [142], a set of rules is described that should be followed in order to develop a programming model for heterogeneous SoCs consisting of multiple programmable cores.

6 Available Design frameworks for DSE

Various design environments have been developed in academia and industry supporting design space exploration of applications and architectures of system and micro-architecture designs. Frameworks mentioned in preceding sections are summarized here to give an overview of existing tools and integrated approaches. A review of tools that are no longer under active development but still cited frequently, such as Polis and VCC, is part of the survey in [143].

We distinguish between tools that are mainly focused on micro-architecture exploration and tools that allow to evaluate a platform on the system level. The former group is mainly based on architecture description languages (ADLs) to ease the generation of retargetable compilers and simulators, whereas the latter group mostly incorporates different design styles and evaluation tools to allow the exploration on the system level. We often find support for SystemC in this group of tools. That means, the system-level centric frameworks can be used to incorporate tools generated by the ADL-centric frameworks with other simulators and models to enable the evaluation of heterogeneous platforms on the system level. Apart from these two classes, we also give examples of other tool flows that may be based on analytical models, architecture templates, or are memory-centric.

6.1 System-level frameworks

Tools in this category allow to model and evaluate architectures and applications on different levels of abstraction using various model descriptions. They therefore often also allow to couple evaluation tools from different sources and vendors to enable the evaluation of heterogeneous architectures on the system level. We often find support for SystemC in these tools to incorporate algorithms and tools written in C and C++. We only list a couple of examples of SystemC-based commercial tools and refer to the Open SystemC Initiative website (*http://www.systemc.org/*) in order to get a comprehensive list of SystemC-based frameworks.

Metropolis [136]. University of California at Berkeley, Politecnico di Torino, and Cadence Berkeley Labs. Metropolis is a framework which allows the description and refinement of a design at different levels of abstraction and integrates modeling, simulation, synthesis, and verification tools. The function of a system, such as the application, is modeled as a set of processes that communicate through media. Non-deterministic behavior can be modeled and constraints can restrict the set of possible executions. Architecture building blocks are represented by performance models where events are annotated with the costs of interest. Further annotations could include arbitrary information, such as a request to access a shared resource, which is subject to centralized control by a corresponding quantity manager. A mapping between functional and architecture models is determined by a third network that correlates the two models by synchronizing events (using constraints) between them.

Mescal [27]. University of California at Berkeley. Mescal is aimed at the design of heterogeneous, application-specific, programmable (multi-) processors. The goal is, on the one hand, to allow the programmer to describe the application in any combination of models of computation that is natural for the application domain, whereas, on the other hand, a disciplined and correct-by-construction abstraction path from the underlying micro-architecture allows an efficient mapping between application and architecture. The architecture development system in Mescal is based on an architecture description language.

StepNP [24]. ST Microelectronics. StepNP is a system-level exploration framework based on SystemC targeted at network processors. It provides well-defined interfaces between processing cores, co-processors, and communication channels to enable the usage of component models at different levels of abstraction. Existing instruction set simulators can be integrated via wrappers that incorporate multi-threaded and pipelined behavior to the overall system simulation. Applications are described in the Click [135] model of computation.

SPADE [12]. Delft University, Leiden University, NL and Philips Research. SPADE implements a trace-driven, system-level co-simulation of application and architecture. Symbolic instruction traces generated by the application are interpreted by architecture models to reveal timing behavior. The application is described by Kahn process networks as provided by YAPI [131]. Abstract, instruction-accurate performance models are used for describing architectures. SPADE is also used in the following work [53][114].

Artemis [14]. Amsterdam, Delft, Leiden Universities, NL, and Philips Research. Artemis is based on a Kahn process network description of the application and incorporates the ideas from SPADE, i.e., system-level co-simulation is performed by using symbolic instruction traces generated and interpreted at run-time by abstract performance models. It adds facilities to explore reconfigurable architectures and for refining architecture models. In [13], an additional layer of virtual processors is introduced between applications and architectures in order to resolve possible deadlocks due to mapping decisions. Artemis is extended to use integer-controlled DDFs in [144] and multiobjective search in [105].

MILAN [17]. University of Southern California and Vanderbilt University. MILAN is a hierarchical design space exploration framework that combines tools for design space pruning with simulators at different levels of abstraction. At the highest level, the design space is expressed symbolically and pruned by constraint satisfaction. Simulators include trace-driven, task-level performance evaluation tools as well as cycle-accurate third-party simulators, such as SimpleScalar [25]. The coarse-level application description follows a kind of hierarchical data flow graph where the behavior of individual nodes is described according to the simulation target, e.g. in Matlab, C, or SystemC.

MESH [145]. Carnegie Mellon University. In MESH, resources (hardware building blocks), software, and schedulers/protocols are seen as three abstraction levels that are modeled by software threads on the evaluation host. Hardware is represented by continuously activated, i.e. rate-based, threads, whereas threads for software and schedulers have no guaranteed activation patterns. The software threads contain annotations describing the hardware requirements, so-called time budgets, that are arbitrated by scheduler threads. Software time budgets are derived beforehand by estimation or profiling. The resolution of a time budget is a design parameter and can thus vary from, e.g., single compute cycles to task-level periods. The advance of simulation time is driven by the periodic hardware threads. The scheduler threads synchronize the available time budgets with the requirements of the software threads.

SEAS [146]. IBM, Corp. The SEAS framework allows the composition of virtual components in order to estimate the performance, area, and power dissipation of the resulting SoC at a high level of abstraction. Each virtual component is associated with a real design component (e.g., specified in VHDL or as a fixed design in a given technology), a performance model described by communicating FSMs, a power state machine describing the transitions between different power states of the architecture, and an area image for floorplaning, representing the size and shape of the design. The software behavior

is described as timed stimuli to the performance and power models, where processing delays are determined by estimation or profiling beforehand. The evaluation in terms of speed and power is done by simulation in System-C. In addition, floorplaning and timing analysis can be done using the abstraction of virtual components.

Incisive-SPW. Coware and Cadence, Inc. The Signal Processing Worksystem (SPW) allows the hierarchical composition of components, supporting synchronous and dynamic data flow models of computation. Components can be taken from Cadence's optimized design library, specified as a state machine, or imported from Matlab, C++, Verilog, VHDL, or SystemC sources. The suggested design flow is based on the iterative refinement of modeling and simulation at multiple levels of abstraction. There is no explicit mapping between application and architecture models [1].

CoCentric System Studio. Synopsys, Inc. System Studio is based on SystemC and thus supports all abstraction levels and models of computation supported by SystemC, such as the hierarchical composition of static and dynamic data flow models, state machines, KPN and TLM models, timed cycle-accurate models, etc. Code import in addition allows to co-simulate HDL descriptions with SystemC models as well as, for instance, Matlab algorithms. Hardware synthesis from SystemC is also supported. Like other SystemC-based tools, System Studio has no explicit mapping step between applications and architectures. The designer implicitly takes mapping decisions by the way he/she interconnects and refines SystemC models which could, for instance, either lead to co-simulation in SystemC or co-simulation of the application and the architecture on a proprietary simulator provided by the manufacturer of the architecture building block.

6.2 Micro-architecture centric frameworks

Tools in this category are mainly focused on the micro-architecture of programmable systems. This is why they are mainly based on architecture description languages (ADLs) in order to describe single processor systems. The ADL descriptions then allow the efficient generation of retargetable tools, such

[1] Cadence's former Virtual Component Co-Design (VCC) framework supported an explicit mapping step where behavioral, executable descriptions were mapped onto architecture building blocks. Depending on the mapping target a binding of a behavior to a certain block could imply a software or hardware implementation. VCC supported different performance model abstractions to accelerate the evaluation of the design under development.

as compiler and simulator. Tools generated by ADL-based frameworks could be coupled and combined with third party tools in one of the system-level frameworks in order to enable the evaluation of a heterogeneous system architecture.

Mescal/Tipi [27,28]. University of California at Berkeley and Infineon Technologies AG. In the architecture development system of Mescal called 'Tipi', the designer only needs to think about the data path of the design. All possible primitive operations that the data path supports are extracted automatically, i.e., the designer does not have to specify any op-codes or control logic elements. The designer can then restrict the set of operations and define more complex instructions from the set of primitive instructions. Cycle-accurate simulators and synthesizable verilog descriptions of the architecture can be generated. The optimization of the memory subsystem can be performed automatically using different optional optimizers and independently of the exploration of the micro-architecture. The micro-architecture description including the memory subsystem is based on an architecture description language.

ASIP-Meister/ PEAS-III [133]. Osaka University. This framework focuses on the development of single programmable processors. Functionality and behavior of the processor and the instruction set are defined in an architecture description language. The language supports traditional micro-architectures by, for instance, having constructs for pipeline stages, delay branch slots, and interrupts. Given an ADL description, simulator, compiler, and VHDL descriptions of the processor are output. Applications can be written in C. The compiler generation is based on the CoSy compiler development system from ACE Associated Compiler Experts bv.

EXPRESSION [32]. University of California at Irvine. EXPRESSION is an architecture description language which enables the modeling of a single programmable processor with its memory subsystem. Simulator, compiler, and VHDL descriptions of the processor can be generated from the EXPRESSION specification of the processor. The application is specified in C++. The description of the micro-architecture is pipeline-centric and the designer specifies all possible, valid data transfers between registers, ALUs, and buses.

LisaTek. CoWare, Inc. Based on a further developed version of the LISA [33] architecture description language for programmable processors, a cycle-accurate simulator, assembler, C-compiler, and debugger can be generated. The feasibility of HDL generation from LISA has been shown in [147]. Possible paths

to multi-processor evaluation are given by CoWare's SystemC-based ConvergenSC System Designer tool, which could potentially couple LISA-generated simulators, or CoWare's LISATek HUB System Integrator multi-processor debugger.

Chess/Checkers. Target Compiler Technologies n.v. Chess/Checkers is an architecture description language-based development system for programmable processors. The processor and the instruction set are described in the nML [34] ADL and the specification may contain multiple memories. Applications are specified in C. Retargetable compilers and instruction set simulators are provided. Synthesizable VHDL models can be generated from the nML description.

MaxCore & MaxSim. Axys Design Automation, Inc. MaxCore is based on the LISA [33] architecture description language and supports the automatic generation of a simulator and an assembler from a LISA processor description. Multi-processor systems can be evaluated by co-simulation of several MaxCore-generated simulators using Axys' MaxSim SoC modeling environment that supports SystemC/C++.

6.3 Related frameworks

In the last category we give examples of other frameworks that integrate further ideas into the exploration process. EXPO is an example for an analytical exploration framework, PICO uses architecture templates to enable the efficient generation of tools, such as the compiler, and lastly Atomium shows a memory-centric exploration method incorporating domain-specific knowledge.

EXPO [128]. Swiss Federal Institute of Technology (ETH) Zurich. EXPO is a system-level, analytical design space exploration tool targeted at applications and SoC architectures in the domain of packet processing. The SoC architecture can be composed of different computation cores, memories, and buses. Each architecture building block is characterized by a non-linear worst-case service curve which represents the capabilities of the resource in, for instance, units of cycles or instructions per second, for all possible time intervals. The application description is an abstract task graph where a sequence of tasks is defined for each traffic flow. Mapping information includes the scheduling policies implemented by each architecture block. In the Intel IXP-specific version of the tool [66], mapping decisions imply certain behavior; e.g., if several tasks within a flow are mapped to the same computation resource, these tasks are

implemented in the same thread of execution and therefore only one of these tasks can be active at a time, etc. Stimuli are described with non-linear arrival curves which are worst-case envelops for all possible traffic patterns. Case studies using EXPO can be found in [65,67,63,64]. The exploration process can be automated using a combination of binary search for the optimization of a single design and multi-objective evolutionary search for the covering of the design space.

PICO [106]. Hewlett-Packard Laboratories. Architectures supported by PICO are composed of one VLIW processing core, a cache hierarchy, and one or several coprocessors. A coprocessor contains a number of functional units organized as a systolic array to accelerate compute-intense loop nests of the original application description. The application input is described in a subset of C. Computing core, caches, and coprocessors are explored independently and Pareto-optimal designs are combined from optimal designs of these subspaces. Architectural choices are bound to configurable templates. The VLIW template, for instance, allows to vary the allocation of functional units and the number of registers. Finally, compiler and simulator as well as synthesizable hardware descriptions can be generated from a PICO design description. Heuristics used in order to explore the design space of the VLIW part of the design are compared in [101] that resemble hill climbing, regular subsampling, and probing with refinement.

Atomium. IMEC, Belgium. Atomium is an exploration tool set for memory subsystems. Given a specification in C, the power consumption of the memory system, the memory size, and the memory speed can be traded-off against each other by code transformations and data layout changes. The techniques used are described in [107] and the references therein.

6.4 Comparison

The features of the mentioned tools are summarized in Table 1 to 3. Tools focused on system-level design allow to express heterogeneous multi-processor systems. The emphasis is put in particular on supporting different abstraction levels and refinement paths. It also often means that SystemC is supported and tools, such as simulators, from different sources can be combined for system-level evaluation. Micro-architecture centric tools usually focus on single, programmable processor systems, where the automatic generation of optimized compilers and simulators becomes important. The following categories are distinguished:

- *Application description:* Representation of the application used by the tool, such as Kahn process networks (KPN), models of computation (MoC), programming languages like C, C++, synchronous or dynamic data flow (SDF and DDF), and finite state machines (FSM).
- *Architecture description:* Architecture representation used, such as performance models, micro-architecture descriptions using architecture description languages (ADL), or hardware description language (HDL) descriptions. The notation 'abstract to HDL' means that the designer has several options and, depending on the chosen architecture representation, the abstraction level could vary from abstract performance models to fine-grained HDL descriptions. ISS stands for instruction set simulation.
- *Exploration modes:* Options for exploring the design space, e.g. automatic vs. manual search in order to evaluate more than one design. 'Script'-based exploration in this context means that the corresponding tools are able to automatically explore designs exhaustively by evaluating all possible permutations of design parameters. Design parameters are exported by, for instance, SystemC modules and represent design decisions, such as clock speed, cache size, and operation bit width. The interconnects, i.e. the topology of the design, cannot be changed automatically by these scripts.
- *Path to hardware:* Here we state whether (synthesizable) models in a hardware description language (HDL) can be introduced into the evaluation. A system-level tool can often combine HDL and other models in a system description so that the whole design can be co-simulated. The HDL models are designed off-line in another tool and imported into the system-level tool. An architecture description language (ADL) based tool however potentially supports the automatic generation of HDL code from the ADL model.
- *Generated tools:* ADL-based tools support the automatic generation of evaluation and development tools, such as simulators ('*sim*'), assemblers ('*asm*'), and compilers ('*comp*').

Please note that we do not distinguish tools regarding their evaluation method. Apart from EXPO [128], which is an analytical framework, all tools are based on simulation and mainly focus on performance. Usually, the precision of the used architecture models determines the precision of the evaluation, e.g. instruction-accurate vs. cycle-accurate. If HDL models can be integrated into the evaluation, further results from synthesis, such as the area consumption, are available.

7 Trade-off analysis

This section discusses trade-offs involved by choosing methods for evaluating a design and for walking through the design space. Information related to this topic can only sparsely be found in related work that deals with automated

Table 1
Feature comparison of design space exploration tools: System-level frameworks.

Name	*Application model*	*Architecture model*	*Exploration mode*	*Path to hardware*	*Tool generation*	*Notes*
Artemis	KPN, DDF	abstract perf. model	scripts, evolutionary optimizer	no	no	Based on SPADE.
CoCentric	FSM, PN, SDF, DDF, Matlab, SystemC	abstract to HDL	manual, scripts	yes	no	
Mescal	mixed MoC	MoC, Tipi	manual	yes (Tipi)	sim	
MESH	threads	abstract perf. model	manual	no	no	
Metropolis	mixed MoC (Meta Model language)	abstract perf. model	manual	(planned)	sim	
MILAN	C, Matlab, Java, SDF	abstract to HDL	manual	yes	no	
SEAS	discrete events, traces	FSMs, abstract to HDL	manual	yes	no	
SPADE	KPN	abstract, instr.-accurate	manual	no	no	
SPW	FSM, C++, Matlab, SystemC	abstract to HDL	manual, scripts	yes	no	
StepNP	Click	abstract, ISS	manual	yes (SystemC)	no	Focus on packet flows.

design space exploration. Most of the argument is therefore based on intuition.

The dimensions of interest are the time to process a single design, the accuracy of the evaluation, and the quality of the design space coverage. More precisely, the time it takes to process one design not only comprises the time required to evaluate (e.g. simulate) it, but also initial efforts to write/program specifications and update requirements of the models and specifications while traversing the design space and remapping applications to architectures.

Quantitative statements on the trade-off concerning evaluation time versus

Table 2
Feature comparison of design space exploration tools: Micro-architecture.

Name	*Application model*	*Architecture model*	*Exploration mode*	*Path to hardware*	*Tool generation*	*Notes*
ASIP-Meister	'C'	ADL	manual	yes (generated)	sim, comp	Uses CoSy compiler from ACE.
Chess/Check.	'C'	nML ADL	manual	yes (generated)	sim, comp	
Expression	C++	Expr. ADL	manual	yes (generated)	sim, comp	
LisaTek	'C'	Lisa ADL	manual	(possible)	sim, comp	
MaxCore	assembler	Lisa ADL	manual	no	sim, asm	
Mescal/Tipi	assembler	ADL-like	partly automatic (memory)	yes (generated)	sim, asm	

Table 3
Feature comparison of design space exploration tools: related frameworks.

Name	*Application model*	*Architecture model*	*Exploration mode*	*Path to hardware*	*Tool generation*	*Notes*
Atomium	'C'	N/A	manual	no	no	Optimization of memory subsystem.
EXPO	DAG, task-level	abstract, accumulated	evolutionary optimizer	no	no	Based on analytical evaluation, system-level.
PICO	'C'	micro-arch. templates	automatic, heuristics	yes (generated)	comp, sim	Templates for VLIW, cache, co-processor.

accuracy can be found in [50] for trace-based analysis. A case study shows a reduction of the evaluation time by two orders of magnitude at the expense of an error of 3.5% in performance compared with a simulation of the complete system.

Quantitative results concerning the quality of covering methods can be found in [101] and [48]. Snider [101] compares three kinds of exploration methods according to the walking/execution time and the area of the design space covered during the walk. The covering methods under investigation resemble hill climbing, probing of the design space with refined exploration around

probes, and regular subsampling of the design space. Fornaciari et al. [48] employ a sensitivity analysis of six design parameters to reduce the number of designs to be evaluated. A case study reveals an exploration demand which is two orders of magnitude smaller than the demand of an exhaustive search. The resulting design shows a energy-delay product that is worse by only 2%.

When we extrapolate those findings to other evaluation methods and determine accuracy versus evaluation time trade-offs, a coarse visualization can be derived in Fig. 10. The figure only considers the evaluation time for one

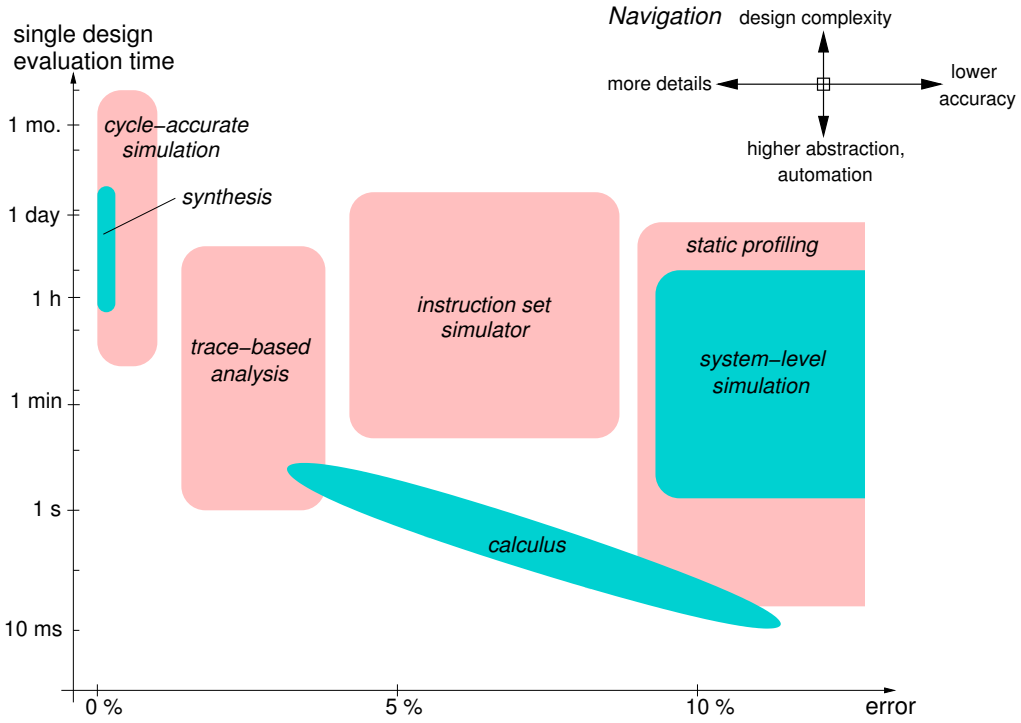

Fig. 10. Evaluation time versus accuracy trade-off for choosing an evaluation method.

design. Initial modeling efforts and updates for exploring the design space are not included. We have chosen cycle-accurate simulation as the reference for DSE evaluation methods. Less accurate evaluation techniques working on higher levels of abstraction consequently allow faster evaluation of a design. We could also walk the sketched trade-off space towards lower accuracy (higher error) by using complexity hiding techniques, such as hierarchical composition. Moreover, the evaluation time increases with increasing design complexity, i.e. higher workloads. Finally, different aspects of the design, such as computation, communication, or control parts, may be modeled on different abstraction levels, effectively blurring the regions in Fig. 10.

The high-level/logic synthesis area in the figure only takes the actual processing time for synthesis into account. Since as a result of the synthesis parameters, such as area consumption or the delay of a schedule, can immediately

be determined, no further simulation runs are required. A co-simulation on the RTL-level would of course require even more time than cycle-accurate simulations. All simulation based approaches include the time for compiling the benchmarks onto the given simulated hardware model. System-level simulation may not require any compilation steps at all. The trade-off area for static profiling is quite stretched, since it includes automated methods, such as function call graph extraction, as well as purely manual profiling phases, such as complexity analysis.

Finally, a qualitative overview of tasks that need to be done in order to initialize an evaluation setup as well as to update a design point from one evaluation to the next during a design space exploration run is shown in Table 4. The

Table 4
Comparison of initialization and update costs for different evaluation methods.

Evaluation method	*Initialization effort*	*Update effort*
Simulation, compiler available	Write application in high-level language.	ADL-based: recompile application.
Simulation, no compiler	Write application in assembler.	Rewrite/update application in assembler.
System-level simulation	Model application and architecture.	Reconfigure model.
Trace-based analysis	Setup and run initial simulation for trace collection.	Trace might require modifications to reflect new setup.
Static profiling	Write application in any language / pseudo-code for analysis.	Rewrite/ update application.
Calculus-based analysis	Specify application and architecture with abstract models.	Modify specification.

work required to initialize and update an evaluation setup could be quite diverse. Some approaches allow automatic updates to the specification, e.g. ADL-, trace-, or calculus-based methods, whereas others require manual operations to be done by the designer. The costs for initialization and update are virtually equal for simulation (no compiler) and static profiling, whereas other methods require a more complex initialization step to ease simple updates, e.g. trace-based analysis. On the one hand, updating a calculus-based analysis can be as simple as running the analysis with a new set of parameters, whereas simulation-based methods might need complete assembler programs adapted to the corresponding architecture.

8 Benchmarking DSE approaches

Although the number of published design space exploration case studies is high, it is difficult to find a common ground among the studies since they often represent isolated results for a particular application domain which lack general comparability and are therefore not indicative of results for other application scenarios. We believe that a systematic benchmarking discipline is required to establish a well-reasoned, quantitative comparison of the quality of design space exploration algorithms. The aim of this section therefore is to point out a possible path to such a benchmarking methodology.

8.1 Indicators for comparing exploration algorithms

We have distinguished algorithms that are used to evaluate a single design point from methods that are employed to walk through the design space. The quality and performance of these algorithms can be assessed by different metrics, as described next.

Algorithms for evaluating a single design point. In this category we are in particular interested in the *accuracy* of the used evaluation method compared to a reference method and the required *computational complexity* to perform the evaluation.

- The *accuracy* could be measured by the deviation from an evaluation of a reference design in terms of primary objectives such as power dissipation and speed.
- The *computational complexity* could be expressed by the time it takes to evaluate a design on a defined evaluation host. This time period should ideally be subdivided into the time spent for specifying and modeling the system under evaluation and the actual evaluation time by, for instance, simulation.

Algorithms for covering the design space. Algorithms for traversing the design space can only meaningfully be assessed in combination with a reference method for evaluating a single design point, e.g. cycle-accurate simulation. In this way, we can be sure to be potentially able to accurately evaluate a reference design if the walking algorithm is capable to discover it. In other words, if a particular covering algorithm cannot find an optimal reference design in the design space, it is due to the properties of the covering algorithm itself and not because of an inferior evaluation method.

The metrics for assessing the quality of a covering algorithm could be quite different depending on the designer's motivation for exploration. On the one hand, if the designer is mainly interested in an unbiased view of the design space so that different regions of the design space could be ranked and the sensitivity on certain design decisions could be figured out, the exposed solutions to the designer are not necessarily Pareto-optimal but represent the variety of possible designs in the problem space. Particularly the following indicators thus become important:

- The *diversity of found designs* in the problem space.
- The *size of the covered space*, be it the problem space or the solution space.

On the other hand, if the designer is mainly interested in quickly finding Pareto-optimal designs, the quality of the set of solutions found by the algorithm under investigation must be compared with the set of solutions of a reference exploration. This comparison of sets is not trivial (see [148] and the references therein). In order to rank different covering algorithms according to their closeness to the reference set of solutions, the assessment metric must carefully be chosen so that the ranking is unambiguous. Zitzler et al. [148] suggest the following two indicators for the pair-wise assessment of sets:

- The *number of solutions weakly dominated (see Def. 3) by solutions of the other set.*
- The *minimum distance between corresponding objectives* of all solutions in both sets (i.e., this is not the Euclidean distance between multiple objectives). The smaller the distance is, the closer the results of the covering algorithm are to the reference set of solutions.

Last but not least, additional computations are required to traverse the design space and decide, which design should be evaluated next. For instance, in the case of evolutionary algorithms, computations are required to calculate the fitness of a solution, to perform mutation and crossover, and to select the next population. Hill-climbing needs to calculate the gradients to neighboring solutions. These computations should therefore also be considered, apart from the computations of the evaluation:

- The *computational complexity* required to decide which design to evaluate next.

Combined evaluation and covering. In order to evaluate the quality of results for a certain combination of an evaluation method with a covering method, additional metrics must be recorded apart from the metrics mentioned before:

- The *overall computational complexity* required to perform the exploration,

e.g. expressed in the execution time on a reference host, includes the evaluation time of all visited designs (including initialization and update costs from one design to the next) and the time it takes to guide the search of the design space.
- The size of the *maximum memory footprint* during the exploration might be one of the interesting properties of the exploration host computer.

Note that the update costs for the evaluation method from one design to the next might tightly depend on the chosen covering algorithm. A guided search may enable the reuse of intermediate evaluation results since the current and the next design could be quite similar, whereas a random search would need to completely remodel the system under evaluation from one design point to the next.

8.2 Defining DSE benchmarks

The purpose of a benchmark is to enable the construction of a repository of comparable, representative, and indicative performance evaluation results. In Section 3.1, we have already listed established benchmarks for evaluating the processing performance of computers and embedded systems. The main goal of those benchmarks is to compare the performance of real systems under a defined environment and a reasonable workload. Part of the benchmark's specification is the application which the architecture under test is supposed to run. Benchmarking reports contain the architecture under evaluation and the maximum achieved performance. We can leverage parts of these established benchmarks to define benchmarks for evaluating design space exploration algorithms.

Since we are primarily interested in comparing DSE algorithms and not the final performance of different designs, not only the application is fixed and part of the specification (like in traditional benchmarks), but also the architecture and the workload must be defined by the benchmark to produce comparable and reproducible results. In concrete terms, the following points need to be considered in order to characterize DSE benchmarks:

- *Specification:* Looking at a particular application domain, such as network processing, a DSE benchmark must contain a description of the application, such as IPv4 forwarding, the constant workload, e.g. 16 bi-directional traffic ports with 100Mb/s throughput and a defined packet length distribution, and one or several descriptions of optimal/reference architectures in the design space, e.g. different micro-architectures of a network processor, as well as a description of the architecture design space to explore, e.g. the number and kind of available memories, buses, peripherals, and processing

elements. The specification of optimal (or at least reference) architectures allows one to reproducibly assess the properties of evaluation methods and the quality of covering algorithms relatively to the set of reference solutions.

- *Representative and reference setups:* For each of the application domains shown in Section 3.1 a representative benchmark should be chosen and adapted to the needs of a DSE benchmark. For embedded systems this could, for instance, mean that an ARM-based architecture is selected to perform a control-dominant application with a defined input from sensors. A couple of ARM-based architectures must be evaluated using a reference evaluation method, such as cycle-accurate simulation, and the performance must be reported. The reference evaluation method as well as the specification and results of the reference architectures now become part of the DSE benchmark.
- *Diversity of exploration domains:* Apart from the application domain DSE benchmarks could also be distinguished according to the main focus of the exploration. A micro-architecture centric exploration should specify different computation styles as reference architectures, such as RISC, DSP, VLIW, ASIP, or even ASIC, in order to investigate whether a certain exploration approach is able to express and exploit this diversity of choices. A system-level centric DSE benchmark should hence focus on different heterogeneous compositions of architecture building blocks.

Constraints on the DSE algorithms themselves, such as bounds on the execution time on a reference evaluation host computer or the size of the memory footprint, should be included in the specification of a DSE benchmark in order to be able to finally answer the following questions for the DSE algorithm under test:

- Are we able to find the reference architectures or even better solutions with respect to defined primary objectives in the ideal case, which does not have any time or resource constraints on the DSE host machine? The quality of the solutions should be reported using the indicators mentioned in the preceding subsection.
- How long does it take/ how much computation is required to find these solutions? What is the maximum memory footprint?
- Given a memory and/or time budget, how good are the achievable results in this case? A series of experiments could trade off the memory consumption against the evaluation time.

The more application domains and abstraction levels are supported by a particular DSE algorithm, the better the confidence in that algorithm should become. This is why it is important to have fundamentally different exploration experiments described in DSE benchmarks.

9 Summary and conclusion

The aim of this paper is to give a comprehensive overview of design space exploration (DSE) techniques used for System-on-a-Chip architectures, including the micro-architecture of single building blocks. We have reviewed academic and commercial frameworks and classified related work according to two orthogonal problem areas: (I) the evaluation of a single design, (II) the representative coverage of the design space. Methods for covering the design space can further be subdivided into methods for searching or pruning the design space, and for automating the exploration process. We have also listed common design space representations and objective functions. We conclude:

- Only evolutionary algorithms or exhaustive search have been used so far in related work to cope with multi-objective optimization design problems. Both approaches are ad-hoc solutions. A more guided search would be desirable to reduce the search complexity. A better incorporation of domain knowledge, e.g. extracted from a coarse characterization of the design space by analytical models, might help to improve convergence to optimal solutions.
- Very little data is available that compares the implementation and evaluation complexity of different DSE approaches. However, the number of published case studies is relatively high. Currently, those studies lack general comparability. This will continue to be a problem as long as well established test cases and benchmarks for DSE are missing.
- Many activities in academia focus on automating mapping and space covering methods, whereas commercial tools ease the manual process of exploring the design space for an experienced designer. The question arises whether an experienced designer will always be able to come up with a suitable solution while facing an increasingly complex design space in the future, or whether automated exploration tools can effectively support the designer in finding feasible designs in this situation.

Consequently, we have drawn up a discipline for benchmarking design space exploration methods and revealed meaningful metrics for this purpose. Defined DSE benchmarks will enable the quantitative and reproducible comparison of design efficiency and quality of results for manual and automatic DSE methods.

Acknowledgements

I would like to thank K. Keutzer, C. Sauer, T. Meyerowitz, C. Kulkarni, and the Mescal team for valuable discussions and comments. This work was

supported, in part, by the Microelectronics Advanced Research Consortium (MARCO) and Infineon Technologies, and is part of the efforts of the Gigascale Systems Research Center (GSRC).

References

[1] N. Shah, Understanding network processors, Master's thesis, Dept. of Electrical Eng. and Computer Sciences, University of California, Berkeley (September 2001).

[2] B. Kienhuis, E. Deprettere, K. Vissers, P. van der Wolf, An approach for quantitative analysis of application-specific dataflow architectures, in: Application-Specific Systems, Architectures, and Processors (ASAP), 1997.

[3] B. De Smedt, G. Gielen, WATSON: a multi-objective design space exploration tool for analog and RF IC design, in: IEEE 2002 Custom Integrated Circuits Conference, 2002, pp. 31–34.

[4] T. Wolf, M. Franklin, CommBench - A telecommunications benchmark for network processors, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2000, pp. 154–162.

[5] G. Memik, W. H. Mangione-Smith, W. Hu, NetBench: A benchmarking suite for network processors, in: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2001.

[6] M. Tsai, C. Kulkarni, C. Sauer, N. Shah, K. Keutzer, A benchmarking methodology for network processors, in: P. Crowley, M. Franklin, H. Hadimioglu, P. Onufryk (Eds.), Network Processor Design: Issues and Practices, Vol. 1, Morgan Kaufmann Publishers, 2002, pp. 141–165.

[7] P. Chandra, F. Hady, R. Yavatkar, T. Bock, M. Cabot, P. Mathew, Benchmarking network processors, in: P. Crowley, M. Franklin, H. Hadimioglu, P. Onufryk (Eds.), Network Processor Design: Issues and Practices, Vol. 1, Morgan Kaufmann Publishers, 2002, pp. 11–25.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: IEEE 4th Annual Workshop on Workload Characterization, 2001, pp. 3–14.

[9] C. Lee, M. Potkonjak, W. Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in: Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997, pp. 330–335.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, in: 22nd International Symposium on Computer Architecture (ISCA), 1995, pp. 24–36.

[11] E. A. Lee, Overview of the Ptolemy project, Tech. Rep. UCB/ERL M03/25, University of California, Berkeley (Jul. 2003).

[12] P. Lieverse, P. van der Wolf, K. Vissers, E. Deprettere, A methodology for architecture exploration of heterogeneous signal processing systems, Kluwer Journal of VLSI Signal Processing 29 (3) (2001) 197–207.

[13] A. Pimentel, S. Polstra, F. Terpstra, A. van Halderen, J. Coffland, L. Hertzberger, Towards efficient design space exploration of heterogeneous embedded media systems, in: Embedded processor design challenges. Systems, architectures, modeling, and simulation - SAMOS, Vol. 2268 of LNCS, Springer-Verlag, 2002, pp. 57–73.

[14] A. Pimentel, L. Hertzberger, P. Lieverse, P. van der Wolf, E. Deprettere, Exploring embedded-systems architectures with Artemis, IEEE Computer 34 (11) (2001) 57–63.

[15] M. Gries, Algorithm-architecture trade-offs in network processor design, Ph.D. thesis, Diss. ETH No. 14191, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland (Jul. 2001).

[16] A. Baghdadi, N. Zergainoh, W. Cesario, T. Roudier, A. Jerraya, Design space exploration for hardware/software codesign of multiprocessor systems, in: 11th International Workshop on Rapid System Prototyping (RSP), 2000, pp. 8–13.

[17] S. Mohanty, V. K. Prasanna, S. Neema, J. Davis, Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation, in: Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2002.

[18] T. Grötker, S. Liao, G. Martin, S. Swan, System Design with SystemC, Kluwer Academic Publishers, 2002.

[19] L. Cai, D. Gajski, M. Olivarez, Introduction of system level architecture exploration using the SpecC methodology, in: IEEE International Symposium on Circuits and Systems, Vol. 5, 2001, pp. 9–12.

[20] L. Cai, S. Verma, D. D. Gajski, Comparison of SpecC and SystemC languages for system design, Tech. Rep. CECS 03-11, University of California, Irvine (May 2003).

[21] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, M. Poncino, SystemC cosimulation and emulation of multiprocessor SoC designs, IEEE Computer 36 (4) (2003) 53–59.

[22] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, S. Goossens, A modular simulation framework for architectural exploration of on-chip interconnection networks, in: CODES/ISSS, 2003.

[23] X. Zhu, S. Malik, Using a communication architecture specification in an application-driven retargetable prototyping platform for distributed processing, in: Design Automation and Test in Europe (DATE), 2004.

[24] P. Paulin, C. Pilkington, E. Bensoudane, StepNP: a system-level exploration platform for network processors, IEEE Design & Test of Computers 19 (6) (2002) 17–26.

[25] D. Burger, T. M. Austin, The SimpleScalar tool set, version 2.0, Tech. Rep. 1342, Computer Sciences Department, University of Wisconsin-Madison (Jun. 1997).

[26] M. Rosenblum, E. Bugnion, S. Devine, S. Herrod, Using the SimOS machine simulator to study complex computer systems, ACM Transactions on Modeling and Computer Simulation 7 (1) (1997) 78–103.

[27] A. Mihal, C. Kulkarni, K. Vissers, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, S. Malik, Developing architectural platforms: A disciplined approach, IEEE Design & Test of Computers 19 (6) (2002) 6–16.

[28] S. J. Weber, M. W. Moskewicz, M. Löw, K. Keutzer, Multi-view operation-level design – supporting the design of irregular ASIPs, Tech. Rep. UCB/ERL M03/12, Electronics Research Laboratory, University of California at Berkeley (Apr. 2003).

[29] P. Mishra, F. Rousseau, N. Dutt, A. Nicolau, Architecture description language driven design space exploration in the presence of co-processors, in: Tenth Workshop on Synthesis And System Integration of MIxed Technologies (SASIMI), 2001.

[30] P. Mishra, N. Dutt, A. Nicolau, Functional abstraction driven design space exploration of heterogeneous programmable architectures, in: International Symposium on System Synthesis, 2001, pp. 256–261.

[31] P. Mishra, F. Rousseau, N. Dutt, A. Nicolau, Coprocessor codesign for programmable architectures, ICS technical report 01-13, Tech. rep., University of California, Irvine (Apr. 2001).

[32] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau, EXPRESSION: A language for architecture exploration through compiler/simulator retargetability, in: Design, Automation and Test in Europe (DATE), 1999, pp. 485–490.

[33] S. Pees, A. Hoffmann, V. Zivojnovic, H. Meyr, LISA-machine description language for cycle-accurate models of programmable DSP architectures, in: 36th Design Automation Conference (DAC), 1999, pp. 933–938.

[34] A. Fauth, J. Van Praet, M. Freericks, Describing instruction set processors using nML, in: European Design and Test Conference (ED&TC), 1995, pp. 503–507.

[35] R. Leupers, P. Marwedel, Retargetable code generation based on structural processor descriptions, Design Automation for Embedded Systems, Kluwer Academic Publishers 3 (1) (1998) 1–36.

[36] E. Schnarr, M. D. Hill, J. R. Larus, Facile: A language and compiler for high-performance processor simulators, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2001, pp. 321–331.

[37] W. Qin, S. Malik, Architecture description languages for retargetable compilation, in: Y. N. Srikant, P. Shankar (Eds.), The Compiler Design Handbook: Optimizations & Machine Code Generation, CRC Press, 2002.

[38] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, A. Nicolau, Architecture description languages for Systems-on-Chip design, in: 6th Asia Pacific Conference on Chip Design Language (APCHDL), 1999, pp. 109–116.

[39] R. Leupers, J. Elste, B. Landwehr, Generation of interpretive and compiled instruction set simulators, in: Asia and South Pacific Design Automation Conference (ASP-DAC), Vol. 1, 1999, pp. 339–342.

[40] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, A. Hoffmann, A universal technique for fast and flexible instruction-set architecture simulation, in: Design Automation Conference (DAC), 2002, pp. 22–27.

[41] M. Reshadi, N. Bansal, P. Mishra, N. Dutt, An efficient retargetable framework for instruction-set simulation, in: CODES/ISSS, 2003.

[42] R. Leupers, P. Marwedel, Retargetable Compiler Technology for Embedded Systems - Tools and Applications, Kluwer Academic Publishers, 2001.

[43] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, H. Meyr, A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA, in: International Conference on Computer Aided Design (ICCAD), San Jose, CA, 2001.

[44] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, G. Goossens, CHESS: Retargetable code generation for embedded DSP processors, in: P. Marwedel, G. Goossens (Eds.), Code Generation for Embedded Processors, Vol. 317 of SECS, Kluwer Academic Publishers, 1995, pp. 85–102.

[45] C. Liem, P. Paulin, Compilation techniques and tools for embedded processor architectures, in: J. Staunstrup, W. Wolf (Eds.), Hardware/Software Co-Design: Principles and Practise, Kluwer Academic Publishers, 1997.

[46] R. Uhlig, T. Mudge, Trace-driven memory simulation: A survey, ACM Computing Surveys 29 (2) (1997) 128–170.

[47] W. Fornaciari, D. Sciuto, C. Silvano, V. Zaccaria, A design framework to efficiently explore energy-delay tradeoffs, in: Ninth International Symposium on Hardware/Software Codesign (CODES), 2001, pp. 260–265.

[48] W. Fornaciari, D. Sciuto, C. Silvano, V. Zaccaria, A sensitivity-based design space exploration methodology for embedded systems, Design Automation for Embedded Systems, Kluwer Academic Publishers 7 (1-2).

[49] T. Givargis, J. Henkel, F. Vahid, Interface and cache power exploration for core-based embedded system design, in: International Conference on Computer-Aided Design (ICCAD), 1999.

[50] K. Lahiri, A. Raghunathan, S. Dey, System-level performance analysis for designing on-chip communication architectures, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20 (6) (2001) 768–783.

[51] K. Lahiri, A. Raghunathan, S. Dey, Efficient exploration of the SoC communication architecture design space, in: IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2000, pp. 424–430.

[52] K. Lahiri, A. Raghunathan, S. Dey, Performance analysis of systems with multi-channel communication architectures, in: Proceedings of 13th International Conference on VLSI Design, 2000, pp. 530–537.

[53] V. Zivkovic, E. Deprettere, P. van der Wolf, E. de Kock, Design space exploration of streaming multiprocessor architectures, in: IEEE Workshop on Signal Processing Systems (SIPS), 2002, pp. 228–234.

[54] M. A. Franklin, T. Wolf, A network processor performance and design model with benchmark parameterization, in: P. Crowley, M. Franklin, H. Hadimioglu, P. Onufryk (Eds.), Network Processor Design: Issues and Practices, Vol. 1, Morgan Kaufmann Publishers, 2002, pp. 117–139.

[55] M. A. Franklin, T. Wolf, Power considerations in network processor design, in: Second Workshop on Network Processors at the 9th International Symposium on High Performance Computer Architecture (HPCA9), 2003.

[56] M. Gries, J. Greutert, Modeling a shared medium access node with QoS distinction, Tech. Rep. 86, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland (Apr. 2000).

[57] D. E. Knuth, The Art of Computer Programming, 3rd Edition, Addison-Wesley, 1997.

[58] R. Sedgewick, Algorithms in Java, Parts 1-4, 3rd Edition, Addison-Wesley, 2002.

[59] Y.-T. S. Li, S. Malik, A. Wolfe, Performance estimation of embedded software with instruction cache modeling, ACM Transactions on Design Automation of Electronic Systems 4 (3) (1999) 257–279.

[60] H. Theiling, C. Ferdinand, R. Wilhelm, Fast and precise WCET prediction by separate cache and path analyses, Real-Time Systems, Kluwer 18 (2-3) (2000) 157–179.

[61] K. Richter, D. Ziegenbein, M. Jersak, R. Ernst, Bottom-up performance analysis of HW/SW platforms, in: Design and Analysis of Distributed Embedded Systems, IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES), Vol. 219 of IFIP Conference Proceedings, Kluwer, Montréal, Québec, Canada, 2002.

[62] J.-Y. Le Boudec, P. Thiran, Network Calculus: A Theory of Deterministic Queuing Systems for the Internet, no. 2050 in LNCS, Springer Verlag, 2001.

[63] L. Thiele, S. Chakraborty, M. Gries, S. Künzli, Design space exploration of network processor architectures, in: P. Crowley, M. Franklin, H. Hadimioglu, P. Onufryk (Eds.), Network Processor Design: Issues and Practices, Vol. 1, Morgan Kaufmann Publishers, 2002, pp. 55–89.

[64] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, J. Greutert, Embedded software in network processors - models and algorithms, in: First Workshop on Embedded Software (EMSOFT), 2001, pp. 416–434.

[65] M. Gries, C. Kulkarni, C. Sauer, K. Keutzer, Exploring trade-offs in performance and programmability of processing element topologies for network processors, in: M. Franklin, P. Crowley, H. Hadimioglu, P. Onufryk (Eds.), Network Processor Design: Issues and Practices, Vol. 2, Morgan Kaufmann, 2003, Ch. 7.

[66] M. Gries, C. Kulkarni, C. Sauer, K. Keutzer, Comparing analytical modeling with simulation for network processors: A case study, in: Design, Automation and Test in Europe (DATE), Munich, Germany, 2003.

[67] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, P. Sagmeister, Performance evaluation of network processor architectures: Combining simulation with analytical estimation, Computer Networks, Elsevier Science 41 (5) (2003) 641–665.

[68] S. Chakraborty, S. Künzli, L. Thiele, A general framework for analysing system properties in platform-based embedded system design, in: Design, Automation and Test in Europe (DATE), Munich, Germany, 2003.

[69] M. Jersak, R. Henia, R. Ernst, Context-aware performance analysis for efficient embedded system design, in: Design, Automation and Test in Europe (DATE), 2004.

[70] A. Maxiaguine, S. Künzli, L. Thiele, Workload characterization model for tasks with variable execution demand, in: Design, Automation and Test in Europe (DATE), 2004.

[71] S. Blythe, R. Walker, Efficiently searching the optimal design space, in: Ninth Great Lakes Symposium on VLSI, IEEE Comput. Soc, 1999, pp. 192–195.

[72] M. Schwiegershausen, P. Pirsch, A system level design methodology for the optimization of heterogeneous multiprocessors, in: Eighth International Symposium on System Synthesis, 1995, pp. 162–167.

[73] S. Blythe, R. Walker, Efficient optimal design space characterization methodologies, ACM Transactions on Design Automation of Electronic Systems 5 (3) (2000) 322–336.

[74] D. S. Rao, F. Kurdahi, Hierarchical design space exploration for a class of digital systems, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 1 (3) (1993) 282–295.

[75] T. Blickle, J. Teich, L. Thiele, System-level synthesis using evolutionary algorithms, Design Automation for Embedded Systems, Kluwer Academic Publishers 3 (1) (1998) 23–58.

[76] K. Chatha, R. Vemuri, An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling, Design Automation for Embedded Systems, Kluwer Academic Publishers 5 (3-4) (2000) 281–293.

[77] I. Ahmad, M. Dhodhi, C. Chen, Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis, IEE Proceedings - Computers and Digital Techniques 142 (1) (1995) 65–71.

[78] R. Dutta, J. Roy, R. Vemuri, Distributed design space exploration for high-level synthesis systems, in: 29th Design Automation Conference (DAC), 1992, pp. 644–650.

[79] R. P. Dick, N. K. Jha, MOCSYN: Multiobjective core-based single-chip system synthesis, in: Design, Automation and Test in Europe Conference (DATE), 1999, pp. 263–270.

[80] A. Ferrari, A. Sangiovanni-Vincentelli, System design: Traditional concepts and new paradigms, in: International Conference on Computer Design (ICCD), 1999, pp. 2–12.

[81] V. Pareto, Cours d'Economie Politique, F.Rouge, Lausanne, 1896.

[82] J. Horn, Multicriterion decision making, in: T. Bäck, D. Forgel, Z. Michalewicz (Eds.), Handbook of Evolutionary Computation, Institute of Physics Publishing, Bristol, UK, 1997.

[83] C. Haubelt, J. Teich, K. Richter, R. Ernst, System design for flexibility, in: Design, Automation and Test in Europe (DATE), 2002, pp. 854–861.

[84] D. Sciuto, F. Salice, L. Pomante, W. Fornaciari, Metrics for design space exploration of heterogeneous multiprocessor embedded systems, in: Tenth International Symposium on Hardware/Software Codesign (CODES), 2002, pp. 55–60.

[85] H. Peixoto, M. Jacome, Algorithm and architecture-level design space exploration using hierarchical data flows, in: IEEE International Conference on Applications-Specific Systems, Architectures and Processors, 1997, pp. 272–282.

[86] C. Ykman-Couvreur, J. Lambrecht, D. Verkest, F. Catthoor, A. Nikologiannis, G. Konstantoulakis, System-level performance optimization of the data queueing memory management in high-speed network processors, in: 39th Design Automation Conference (DAC), 2002.

[87] S. Blythe, R. Walker, Toward a practical methodology for completely characterizing the optimal design space, in: 9th International Symposium on System Synthesis, 1996, pp. 8–13.

[88] S. Chaudhuri, S. Blythe, R. Walker, A solution methodology for exact design space exploration in a three-dimensional design space, in: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 5, 1997, pp. 69–81.

[89] M. Auguin, L. Capella, F. Cuesta, E. Gresset, CODEF: a system level design space exploration tool, in: 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 2, 2001, pp. 1145–1148.

[90] R. Szymanek, K. Kuchcinski, Design space exploration in system level synthesis under memory constraints, in: 25th EUROMICRO Conference, Vol. 1, 1999, pp. 29–36.

[91] S. Pees, A. Hoffmann, H. Meyr, Retargeting of compiled simulators for digital signal processors using a machine description language, in: Design, Automation and Test in Europe Conference (DATE), 2000, pp. 669–673.

[92] J. Kin, C. Lee, W. Mangione-Smith, M. Potkonjak, Power efficient mediaprocessors: design space exploration, in: 36th Design Automation Conference (DAC), 1999, pp. 321–326.

[93] G. Hekstra, G. L. Hei, P. Bingley, F. Sijstermans, TriMedia CPU64 design space exploration, in: 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1999, pp. 599–606.

[94] I. Karkowski, H. Corporaal, Design space exploration algorithm for heterogeneous multi-processor embedded system design, in: 35th Design and Automation Conference (DAC), 1998, pp. 82–87.

[95] D. Bruni, A. B. L. Benini, Statistical design space exploration for application-specific unit synthesis, in: 38th Design Automation Conference (DAC), 2001, pp. 641–646.

[96] V. Srinivasan, S. Radhakrishnan, R. Vemuri, Hardware software partitioning with integrated hardware design space exploration, in: Design, Automation and Test in Europe (DATE), 1998, pp. 28–35.

[97] D. Gajski, F. Vahid, S. Narayan, J. Gong, System-level exploration with SpecSyn, in: 35th Design and Automation Conference (DAC), 1998, pp. 812–817.

[98] F. Glover, M. Laguna, Tabu Search, Kluwer Academic Publishers, 1997.

[99] F. Moya, J. Moya, J. Lopez, Evaluation of design space exploration strategies, in: 25th EUROMICRO Conference, Vol. 1, 1999, pp. 472–476.

[100] J. Axelsson, Architecture synthesis and partitioning of real-time systems: a comparison of three heuristic search strategies, in: 5th Int. Workshop on Hardware/Software Codesign (CODES/CASHE), 1997, pp. 161–165.

[101] G. Snider, Spacewalker: Automated design space exploration for embedded computer systems, HPL-2001-220, Tech. rep., HP Laboratories Palo Alto (Sep. 2001).

[102] M. Palesi, T. Givargis, Multi-objective design space exploration using genetic algorithms, in: Tenth International Symposium on Hardware/Software Codesign (CODES), 2002, pp. 67–72.

[103] G. Ascia, V. Catania, M. Palesi, Design space exploration methodologies for IP-based system-on-a-chip, in: IEEE International Symposium on Circuits and Systems, Vol. 2, 2002, pp. 364–367.

[104] G. Ascia, V. Catania, M. Palesi, A framework for design space exploration of parameterized VLSI systems, in: ASP-DAC/VLSI Design 2002, 2002, pp. 245–250.

[105] C. Erbas, S. C. Erbas, A. D. Pimentel, A multiobjective optimization model for exploring multiprocessor mappings of process networks, in: CODES/ISSS, 2003.

[106] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. Cronquist, M. Sivaraman, PICO: automatically designing custom computers, IEEE Computer 35 (9) (2002) 39–47.

[107] M. Miranda, C. Ghez, C. Kulkarni, F. Catthoor, D. Verkest, Systematic speed-power memory data-layout exploration for cache controlled embedded multimedia applications, in: International Symposium on System Synthesis, 2001, pp. 107–112.

[108] C. Kulkarni, D. Moolenaar, L. Nachtergaele, F. Catthoor, H. De Man, System level energy-delay exploration for multimedia applications on embedded cores with hardware caches, Kluwer Journal of VLSI Signal Processing 22 (1) (1999) 45–57.

[109] T. Givargis, F. Vahid, J. Henkel, System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 10 (4) (2002) 416–422.

[110] V. Mathur, V. Prasanna, A hierarchical simulation framework for application development on system-on-chip architectures, in: 14th Annual IEEE International ASIC/SOC Conference, 2001, pp. 428–434.

[111] H. Hsieh, F. Balarim, L. Lavagno, A. Sangiovanni-Vincentelli, Efficient methods for embedded system design space exploration, in: 37th Design Automation Conference (DAC), 2000, pp. 607–612.

[112] C. Wilson, D. L. Dill, R. E. Bryant, Symbolic simulation with approximate values, in: Third International Conference on Formal Methods in Computer-Aided Design, 2000.

[113] J. Peng, S. A. D. Gajski, Automatic model refinement for fast architecture exploration, in: ASP-DAC/VLSI Design 2002, 2002, pp. 332–337.

[114] P. Lieverse, P. van der Wolf, E. Deprettere, A trace transformation technique for communication refinement, in: Ninth International Symposium on Hardware/Software Codesign (CODES), 2001, pp. 134–139.

[115] L. Eeckhout, K. de Bosschere, H. Neefs, Performance analysis through synthetic trace generation, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2000, pp. 1–6.

[116] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, A. Jerraya, A generic wrapper architecture for multi-processor SoC cosimulation and design, in: 9th International Symposium on Hardware/Software Codesign (CODES), 2001, pp. 195–200.

[117] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli, Addressing the System-on-a-Chip interconnect woes through communication-based design, in: Design Automation Conference (DAC), 2001, pp. 667–672.

[118] B. Lin, S. Vercauteren, Synthesis of concurrent system interface modules with automatic protocol conversion generation, in: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 1994, pp. 101–108.

[119] J. Smith, G. DeMicheli, Automated composition of hardware components, in: Design Automation Conference (DAC), 1998, pp. 14–19.

[120] R. Passerone, J. A. Rowson, A. Sangiovanni-Vincentelli, Automatic synthesis of interfaces between incompatible protocols, in: Design Automation Conference (DAC), 1998, pp. 8–13.

[121] R. B. Ortega, G. Borriello, Communication synthesis for distributed embedded systems, in: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 1998, pp. 437–444.

[122] K. Richter, D. Ziegenbein, M. Jersak, R. Ernst, Model composition for scheduling analysis in platform design, in: Design Automation Conference (DAC), 2002, pp. 287–292.

[123] M. O'Nils, A. Jantsch, Operating system sensitive device driver synthesis from implementation independent protocol specification, in: Design, Automation and Test in Europe (DATE), 1999, pp. 562–567.

[124] S. Wang, S. Malik, R. Bergamaschi, Modeling and integration of peripheral devices in embedded systems, in: Design, Automation and Test in Europe (DATE), 2003, pp. 136–141.

[125] L. Gauthier, S. Yoo, A. Jerraya, Automatic generation and targeting of application specific operating systems and embedded systems software, in: Design, Automation and Test in Europe (DATE), 2001, pp. 679–685.

[126] F. Balarin, M. Chiodo, A. Jurecska, L. Lavagno, B. Tabbara, A. Sangiovanni-Vincentelli, Automatic generation of a real-time operating system for embedded systems, in: 5th International Workshop on Hardware/Software Co-Design (Codes/CASHE), 1997.

[127] X. Hu, G. Greenwood, S. Ravichandran, G. Quan, A framework for user assisted design space exploration, in: 36th Design Automation Conference (DAC), 1999, pp. 414–419.

[128] L. Thiele, S. Chakraborty, M. Gries, S. Künzli, A framework for evaluating design tradeoffs in packet processing architectures, in: 39th Design Automation Conference (DAC), New Orleans LA, USA, 2002, pp. 880–885.

[129] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, D. I. August, Microarchitectural exploration with Liberty, in: 35th International Symposium on Microarchitecture (MICRO), 2002, pp. 271–282.

[130] G. Kahn, The semantics of a simple language for parallel programming, in: Proceedings of the IFIP Congress, North-Holland Publishing Co., 1974, pp. 471–475.

[131] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, YAPI: Application modeling for signal processing systems, in: 37th Design Automation Conference (DAC), 2000, pp. 402–405.

[132] T. Harriss, R. Walke, B. Kienhuis, E. Deprettere, Compilation from Matlab to process networks realized in FPGA, Design Automation for Embedded Systems, Kluwer 7 (4) (2002) 385–403.

[133] S. Kobayashi, K. Mita, Y. Takeuchi, M. Imai, A compiler generation method for HW/SW codesign based on configurable processors, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E85-A (12) (2002) 2586–2595.

[134] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara, Hardware-Software Co-Design of Embedded Systems: The Polis Approach, no. 404 in International Series in Engineering and Computer Science, Kluwer Academic Publishers, 1997.

[135] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek, The Click modular router, ACM Transactions on Computer Systems 18 (3) (2000) 263–297.

[136] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Paserone, A. Sangiovanni-Vincentelli, Metropolis: an integrated electronic system design environment, IEEE Computer 36 (4) (2003) 45–52.

[137] N. Shah, W. Plishker, K. Keutzer, NP-Click: A programming model for the Intel IXP1200, in: M. Franklin, P. Crowley, H. Hadimioglu, P. Onufryk (Eds.), Network Processor Design: Issues and Practices, Vol. 2, Morgan Kaufmann, 2003, Ch. 9.

[138] Intel Corp., Introduction to the auto-partitioning programming model, White Paper (Oct. 2003).

[139] J. L. Pino, S. Ha, E. A. Lee, J. T. Buck, Software synthesis for DSP using Ptolemy, VLSI Signal Processing 9 (1) (1995) 7–21.

[140] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, J. D. Owens, Programmable stream processors, IEEE Computer 36 (8) (2003) 54–62.

[141] W. Gropp, E. Lusk, A taxonomy of programming models for symmetric multiprocessors and SMP clusters, in: Programming Models for Massively Parallel Computers, 1995, pp. 2–7.

[142] J. M. Paul, Programmer's views of SoCs, in: CODES/ISSS, 2003.

[143] V. D. Zivkovic, P. Lieverse, An overview of methodologies and tools in the field of system-level design, in: E. F. Deprettere, J. Teich, S. Vassiliadis (Eds.), Embedded Processor Design Challenges: 2nd International Samos Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS), no. 2268 in LNCS, Springer Verlag, 2002, pp. 74–88.

[144] C. Erbas, A. Pimentel, Utilizing synthesis methods in accurate system-level exploration of heterogeneous embedded systems, in: IEEE Workshop on Signal Processing Systems (SIPS), 2003.

[145] A. Cassidy, J. Paul, D. Thomas, Layered, multi-threaded, high-level performance design, in: Design, Automation and Test in Europe (DATE), 2003, pp. 954–959.

[146] R. A. Bergamaschi, Y. Shin, N. Dhanwada, S. Bhattacharya, W. Dougherty, I. Nair, J. Darringer, S. Paliwal, SEAS: A system for early analysis of SoCs, in: CODES/ISSS, 2003.

[147] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, H. Meyr, Architecture implementation using the machine description language LISA, in: 15th International Conference on VLSI Design, 2002, pp. 239–244.

[148] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, V. G. da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, IEEE Transactions on Evolutionary Computation 7 (2) (2003) 117–132.

Matthias Gries is a post-doctoral researcher at the University of California, Berkeley, in the Computer-Aided Design group. He received the Doctor of Technical Sciences degree from the Swiss Federal Institute of Technology (ETH) Zürich in 2001 for his work on the system-level design of a QoS network processor. He received the Dipl.-Ing. degree in electrical engineering from the Technical University Hamburg-Harburg, Germany, in 1996, working on reconfigurable architectures for video processing. His interests include methods and tools for developing application-specific processors, system-level design, and analysis of real-time embedded systems.