CHAPTER 1

A COMPUTER ENGINEERING BENCHMARK APPLICATION FOR MULTIOBJECTIVE OPTIMIZERS

Simon Künzli, Stefan Bleuler, Lothar Thiele, and Eckart Zitzler

Department of Information Technology and Electrical Engineering
Swiss Federal Institute of Technology (ETH) Zurich
Gloriastrasse 35, CH-8092 Zurich, Switzerland
E-mail: {kuenzli,bleuler,thiele,zitzler}@tik.ee.ethz.ch

Among the various benchmark problems designed to compare and evaluate the performance of multiobjective optimizers, there is a lack of real-world applications that are commonly accepted and, even more important, are easy to use by different research groups. The main reason is, in our opinion, the high effort required to re-implement or adapt the corresponding programs.

This chapter addresses this problem by presenting a demanding packet processor application with a platform and programming language independent interface. The text-based interface has two advantages: it allows (i) to distribute the application as a binary executable pre-compiled for different platforms, and (ii) to easily couple the application with arbitrary optimization methods without any modifications on the application side. Furthermore, the design space exploration application presented here is a complex optimization problem that is representative for many other computer engineering applications. For these reasons, it can serve as a computer engineering benchmark application for multiobjective optimizers. The program can be downloaded together with different multiobjective evolutionary algorithms and further benchmark problems from *http://www.tik.ee.ethz.ch/pisa/*.

1. Introduction

The field of evolutionary multiobjective optimization (EMO) has been growing rapidly since the first pioneering works in the mid-1980's and the early 1990's. Meanwhile numerous methods and algorithmic components are available, and accordingly there is a need for representative benchmark problems to compare and evaluate the different techniques.

Most test problems that have been suggested in the literature are artificial and abstract from real-world scenarios. Some authors considered multiobjective extensions of NP-hard problems such as the knapsack problem [27], the set covering problem [13], and the quadratic assignment problem [14]. Other benchmark problem examples are the Pseudo-Boolean functions introduced by Thierens [23] and Laumanns et al. [15] that were designed mainly for theoretical investigations. Most popular, though, are real-valued functions [7,6]. For instance, several test functions representing different types of problem difficulties were proposed by Zitzler et al. [24] and Deb et al. [9].

Although there exists no commonly accepted set of benchmark problems as, e.g., the SPEC benchmarks in computer engineering, most of the aforementioned functions are used by different researchers within the EMO community. The reason is that the corresponding problem formulations are simple which in turn keeps the implementation effort low. However, the simplicity and the high abstraction level come along with a loss of information: various features and characteristics of real-world applications cannot be captured by these artificial optimization problems. As a consequence, one has to test algorithms also on actual applications in order to obtain more reliable results. Complex problems in various areas have been tackled using multiobjective evolutionary algorithms, and many studies even compare two or several algorithms on a specific application [7,6]. The restricted reusability, though, has prohibited so far that one or several applications have established themselves as benchmark problems that are used by *different* research groups. Re-implementation is usually too labor-intensive and error-prone, while re-compilation is often not possible because either the source code is not available, e.g., due to intellectual property issues, or particular software packages are needed that are not publicly available.

To solve this problem, we here present a computer engineering application , namely the design space exploration of packet processor architectures, that

- provides a platform and programming language independent interface that allows the usage of pre-compiled and executable programs and therefore circumvents the problem mentioned above,
- is scalable in terms of complexity, i.e., problem instances of different levels of difficulty are available, and
- is representative for several other applications in the area of computer design [3,10,26].

The application will be described in terms of the underlying optimiza-

tion model in Section 2, while Section 3 focuses on the overall software architecture and in particular on the interface. Section 4 demonstrates the application of four EMO techniques on several problem instances and compares their performance on the basis of a recently proposed quality measure [28]. The last section summarizes the main results of this chapter.

2. Packet Processor Design

Packet processores are high-performance, programmable devices with special architectural features that are optimized for network packet processing.[a] They are mostly embedded within network routers and switches and are designed to implement complex packet processing tasks at high line speeds such as routing and forwarding, firewalls, network address translators, means for implementing quality-of-service (QoS) guarantees to different packet flows, and also pricing mechanisms.

Other examples of packet processors would be media processors which have network interfaces. Such processors have audio, video and packet-processing capabilities and serve as a bridge between a network and a source/sink audio/video device. They are used to distribute (real-time) multimedia streams over a packet network like wired or wireless Ethernet. This involves receiving packets from a network, followed by processing in the protocol stack, forwarding to different audio/video devices and applying functions like decryption and decompression of multimedia streams. Similarly, at source end, this involves receiving multimedia streams from audio/video devices (e.g. video camera, microphone, stereo systems), probably encrypting, compressing and packetizing them, and finally sending them over a network.

Following the above discussion, there are major constraints to satisfy and conflicting goals to optimize in the design of packet processors:

- **Delay Constraints:** In case of packets belonging to a multimedia stream, there is very often a constraint on the maximal time a packet is allowed to stay within the packet processor. This upper delay must be satisfied under all possible load conditions imposed by other packet streams that are processed simultaneously by the same device.
- **Throughput Maximization:**It is the goal to maximize the maximum possible throughput of the packet processing device in terms

[a]In this area of application, also the term network processor is used.

of the number of packets per second.

- **Cost Minimization:** One is interested in a design that uses a small amount of resources, e.g. single processing units, memory and communication networks.
- **Conflicting Usage Scenarios:** Usually, a packet processor is used in several different systems. For example, one processor will be implemented within a router, another one is built into a consumer device for multimedia processing. The requirements from these different applications in terms of throughput and delay are typically in conflict to each other.

All of the above constraints and conflicting goals will be taken into account in the benchmark application.

2.1. *Design Space Exploration*

Complex embedded systems like packet processors are often comprised of a heterogeneous combination of different hardware and software components such as CPU cores, dedicated hardware blocks, different kinds of memory modules and caches, various interconnections and I/O interfaces, run-time environment and drivers, see e.g. Figure 1. They are integrated on a single chip and they run specialized software to perform the application.

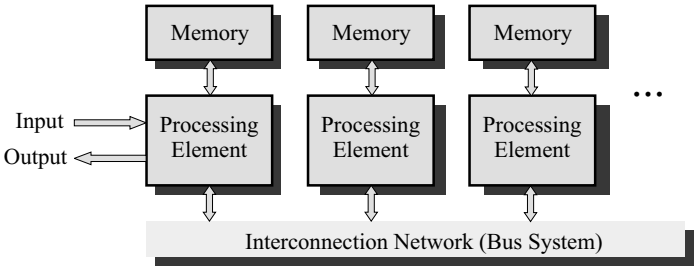

Fig. 1. Template of a packet processor architecture as used in the benchmark application.

Typically, the analysis questions faced by a designer during a system-level design process are:

- **Allocation:** Determine the hardware components of the packet processor, like microprocessors, dedicated hardware blocks for computation intensive application tasks, memory and busses.

- **Binding :** Choose for each task of the software application an allocated hardware unit which executes it.
- **Scheduling Policy:** Choose for the set of tasks that are mapped onto a specific hardware resource a scheduling policy from the available run-time environment, e.g. a fixed priority.

Most of the available design methodologies start with an abstract specification of the application and the performance requirements. These specifications are used to drive a system-level design space exploration [17], which iterates between performance evaluation and exploration steps, see also [19], [20], [3]. Finally, appropriate allocations, bindings and scheduling strategies are identified. The methodology used in the benchmark application of this paper is shown in Figure 2.

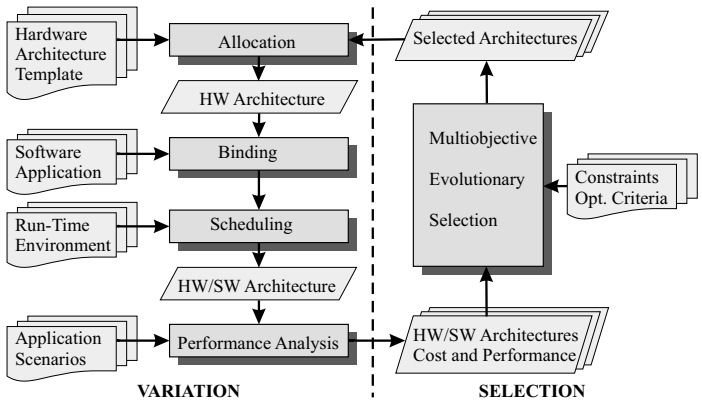

Fig. 2. Design space exploration methodology used in the benchmark application.

One of the major challenges in the design exploration is to estimate the essential characteristics of the final implementation in an early design stage. In case of the packet processor design, the performance analysis has to cope with two major problems which make any kind of compositional analysis difficult: (1) The architecture of such systems is highly heterogeneous—the different architectural components have different computing capabilities and use different arbitration and resource sharing strategies. (2) The packets of one or different packet streams interact on the various resources, i.e. if a resource is busy in processing one packet, others have to wait. This interaction between packet streams is of a tremendous complexity and in-

fluences packet delays and memory usage.

There is a large body of work devoted to system-level performance analysis of embedded system architectures, see [12] and the references therein. Currently, the analysis of such heterogeneous systems is mainly based on simulation. The main advantage of using simulation as a means for performance evaluation is that many dynamic and complex interactions in an architecture can be taken into account, which are otherwise difficult to model analytically. On the other hand, simulation based tools suffer from high running times, incomplete coverage, and failure to identify corner cases.

Analytical performance models for DSP systems and embedded processors were proposed in e.g. [1,11]. These models may be classified under what can be called a "static analytical model". Here, the computation, communication, and memory resources of a processor are all described using simple algebraic equations that do not take into account the dynamics of the application, i.e. variations in resource loads and shared resources. In contrast to this class of approaches, the models we will use in the paper may be classified under "dynamic analytical models", where the dynamic behavior of the computation and communication resources (such as the effects of different scheduling or bus arbitration schemes) are also modeled, see e.g. [16,22,4]. Applications to stream-processing can be found in [21,18,19,5].

2.2. *Basic Models and Methods*

According to Figure 2, basic prerequisites of the design space exploration are models for the architecture, the application, the run-time scheduling, and the application scenarios. Based on these models, we will describe the method for performance analysis.

Architecture Template and Allocation Following Figure 1, the model for a packet processor consists of a set of computation units or processing elements which perform operations on the individual packets. In order to simplify the discussion and the benchmark application, we will not model the communication between the processing elements, i.e. packets can be moved from one memory element to the next one without constraints.

Definition 1: We define a set of resources R. To each resource $r \in R$ we associate a relative implementation cost $cost(r) \geq 0$. The allocation of resources is described by the function $alloc(r) \in \{0, 1\}$. To each resource r there are associated two functions $\beta_r^u(\Delta) \geq 0$ and $\beta_r^l(\Delta) \geq 0$, denoted as upper and lower service curves, respectively.

Initially, we specify all *available* processing units as our resource set R and associate the corresponding costs to them. For example we may have the resources R = {ARM9, MEngine, Classifier, DSP, Cipher, LookUp, CheckSum, PowerPC}. During the allocation step (see Figure 2), we select those which will be in a specific architecture, i.e. if $alloc(r) = 1$, then resource $r \in R$ will be implemented in the packet processor architecture.

The upper and lower service curves specify the available *computing units* of a resource r in a relative measure, e.g. processor cycles or instructions. In particular, $\beta_r^u(\Delta)$ and $\beta_r^l(\Delta)$ are the maximum and minimum number of available processor cycles in any time interval of length Δ. In other words, the service curves of a resource determine the best case and worst case computing capabilities. For details, see e.g. [18].

Software Application and Binding The purpose of a packet processor is to simultaneously process several streams of packets. For example, one stream may contain packets that store audio samples and another one contains packets from an ftp application. Whereas the different streams may be processed differently, each packet of a particular stream is processed identically, i.e. each packet is processed by the same sequence of tasks.

Definition 2: We define a set of streams $s \in S$ and a set of tasks $t \in T$. To each stream s there is associated an ordered sequence of tasks $V(s) = [t_0, ..., t_n]$. Each packet of the stream is first processed by task $t_0 \in T$, then successively by all other tasks until $t_n \in T$.

As an example we may have five streams F = {RTSend, NRTDecrypt, NRTEncrypt, RTRecv, NRTForward}. According to Figure 3, the packets of these streams when entering the packet processor undergo different sequences of tasks, i.e. the packets follow the path shown. For example, for stream s = NRTForward we have the sequence of tasks $V(s)$ = [LinkRX, VerifyIPHeader, ProcessIPHeader, Classify, RoutLookUp, ... , Schedule, LinkTx].

Definition 3: The mapping relation $M \subseteq T \times R$ defines all possible bindings of tasks, i.e. if $(t, r) \in M$, then task t could be executed on resource r. This execution of t for one packet would use $w(r, t) \geq 0$ computing units of r. The binding B of tasks to resources $B \subseteq M$ is a subset of the mapping such that every task $t \in T$ is bound to exactly one allocated resource $r \in R$, $alloc(r) = 1$. We also write $r = bind(t)$ in a functional notation.

In a similar way as *alloc* describes the selection of architectural com-

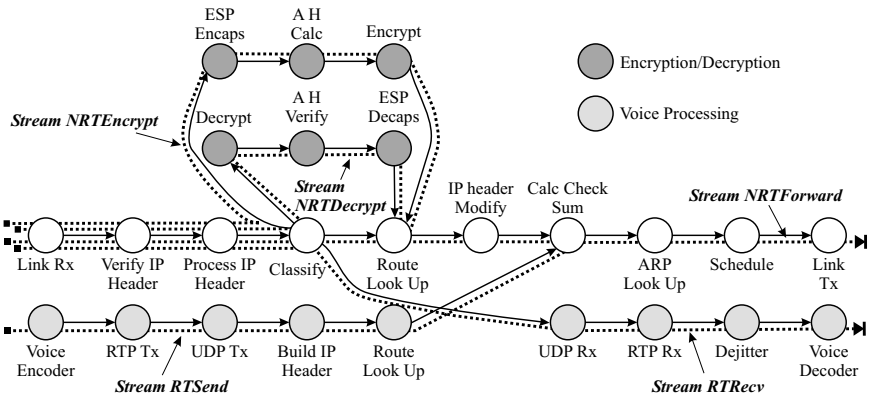

Fig. 3. Task graph of a packet processing application.

ponents, *bind* defines a selection of the possible mappings. Both *alloc* and *bind* will be encoded using an appropriate representation described later. The 'load' that a task t puts onto its resource $r = bind(t)$ is denoted as $w(r, t)$.

Figure 4 represents an example of a mapping between tasks and resources. For example, task 'Classify' could be bound to resource 'ARM9' or 'DSP'. In a particular implementation of a packet processor we may have $bind(\text{Classify}) = \text{DSP}$, i.e. the task 'Classify' is executed on the resource 'DSP' and the corresponding execution requirement for each packet is $w(\text{DSP}, \text{Classify}) = 2.9$. Of course, this is possible only, if the resource is allocated, i.e. $alloc(\text{DSP}) = 1$.

Run-time Environment and Scheduling According to Figure 1, there is a memory associated to each processing element. Within this memory, all packets are stored that need to be processed by the respective resource. The run-time environment now has different scheduling policies available that determine which of the waiting packets will be processed next.

Definition 4: To each stream s there is associated an integer priority $prio(s) > 0$. There are no streams with equal priority.

In the benchmark application, we suppose that only preemptive fixed-priority scheduling is available on each resource. To this end, we need to associate to each stream s a fixed priority $prio(s) > 0$, i.e. all packets of s receive this priority. From all packets that wait to be executed in a memory, the run-time environment chooses one for processing that has the highest

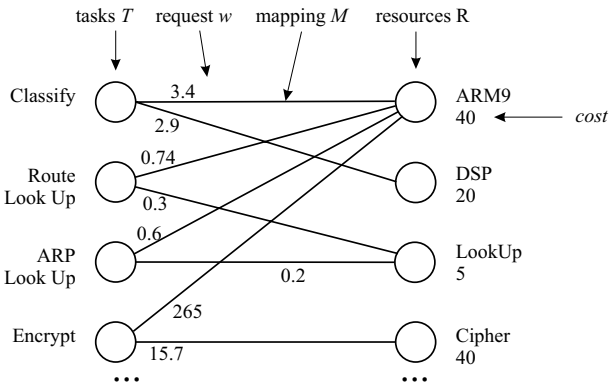

Fig. 4. Example of a mapping of task to recourses.

priority among all waiting packets. If several packets from one stream are waiting, then it prefers those that are earlier in the task chain $V(s)$.

Application Scenarios A packet processor will be used in several, possibly conflicting application scenarios. Such a scenario is described by the properties of the input streams, the allowable end-to-end delay (deadline) for each stream and the available total memory for all packets (sum of all individual memories of the processing elements).

Definition 5: The properties of each stream s are described by upper and lower arrival curves $\alpha_s^u(\Delta)$ and $\alpha_s^l(\Delta)$. To each stream $s \in S$ there is associated the maximal total packet memory $m(s) \geq 0$ and an end-to-end deadline $d(s) \geq 0$, denoting the maximal time by which any packet of the stream has to be processed by all associated tasks $V(s)$ after his arrival.

The upper and lower service arrival specify upper and lower bounds on the number of packets that arrive at the packet processor. In particular, $\alpha_s^u(\Delta)$ and $\beta_s^l(\Delta)$ are the maximum and minimum number of packets in *any time interval* of length Δ. For details, see e.g. [21].

Definition 6: The packet processor is evaluated for a set of scenarios $b \in B$. The quantities of Definition 5 are defined for each scenario independently.

In addition, whereas the allocation *alloc* is defining a particular hardware architecture, the quantities that are specific for a software application

are also specific for each scenario $b \in B$ and must be determined independently, for example the binding *bind* of tasks to processing elements and the stream priorities *prio*.

Performance Analysis It is not obvious how to determine for any memory module, the maximum number of stored packets in it waiting to be processed at any point in time. Neither is it clear how to determine the maximum end-to-end delays experienced by the packets, since all packet flows share common resources. As the packets may flow from one resource to the next one, there may be intermediate bursts and packet jams, making the computations of the packet delays and the memory requirements non-trivial.

Interestingly, there exists a computationally efficient method to derive worst-case estimates on the end-to-end delays of packets and the required memory for each computation and communication. In short, we construct a scheduling network and apply the real-time calculus (based on arrival and service curves) in order to derive the desired bounds. The description of this method is beyond the scope of this chapter but can be found in [21,18,19].

As we know for each scenario the delay and memory in comparison to the allowed values $d(b,s)$ and $m(b,s)$, we can increase the input traffic until the constraints are just satisfied. In particular, we do not use the arrival curves $\alpha^u_{(b,s)}$ and $\alpha^l_{(b,s)}$ directly in the scheduling network, but linearly scaled amounts $\psi_b \cdot \alpha^u_{(b,s)}$ and $\psi_b \cdot \alpha^l_{(b,s)}$, where the scaling factor ψ_b is different for each scenario. Now, binary search is applied to determine the maximal throughput such that the constraints on delay and memory are just satisfied.

For the following discussion, it is sufficient to state the following fact:

- Given the specification of a packet processing design problem by the set of resources $r \in R$, the cost function for each resource $cost(r)$, the service curves β^u_r and β^l_r, a set of streams $s \in S$, a set of application tasks $t \in T$, the ordered sequence of tasks for each stream $V(s)$, and the computing requirement $w(r,t)$ for task t on resource r.
- Given a set of application scenarios $b \in B$ with associated arrival curves for each stream $\alpha^u_{(b,s)}$ and $\alpha^l_{b,s}$, and a maximum delay and memory for each stream $d(b,s)$ and $m(b,s)$.
- Given a specific HW/SW architecture defined by the allocation of hardware resources $alloc(r)$, for each scenario b a specific priority

of each stream $prio(b, s)$ and a specific binding $bind(b, t)$ of tasks t to resources.

- Then we can determine using the concepts of scheduling network, real-time calculus and binary search the maximal scaling factor ψ_b such that under the input arrival curves $\psi_b \cdot \alpha^u_{(b,s)}$ and $\psi_b \cdot \alpha^l_{b,s}$ the maximal delay of each packet and the maximal number of stored packets is not larger than $d(b, s)$ and $m(b, s)$, respectively.

As a result, we can define the criteria for the optimization of packet processors.

Definition 7: The quality measures for packet processors are the associated cost $cost = \sum_{r \in R} alloc(r)cost(r)$ and the throughput ψ_b for each scenario $b \in B$. These quantities can be computed from the specification of a HW/SW architecture, i.e. $alloc(r)$, $prio(b, s)$ and $bind(b, t)$ for all streams $s \in S$ and tasks $t \in T$.

Representation Following Figure 2 and Definition 7, a specific HW/SW architecture is defined by $alloc(r)$, $prio(b, s)$ and $bind(b, t)$ for all resources $r \in R$, streams $s \in S$ and tasks $t \in T$. For the representation of architectures, we number the available resources from 1 to $|R|$; the tasks are numbered from 1 to $|T|$, and each stream is assigned a number between 1 and $|S|$. The allocation of resources can then be represented as integer vector $A \in \{0, 1\}^{|R|}$, where $A[i] = 1$ denotes, that resource i is allocated. To represent the binding of tasks on resources, we use a two-dimensional vector $Z \in \{1, \ldots, |R|\}^{|B| \times |T|}$, where for all scenarios $b \in B$ it is stored which task is bound to which resource. $Z[i][j] = k$ means that in scenario i task j is bound to resource k. Priorities of flows are represented as a two-dimensional vector $P \in \{1, \ldots, |S|\}^{|B| \times |S|}$, where we store the streams according to their priorities, e. g. $P[i][j] = k$ means that in scenario i, stream k has priority j, with 1 being the highest priority. Obviously, not all possible encodings A, Z, P represent feasible architectures. Therefore, a repair method exists that converts infeasible solutions into feasible ones.

Recombination The first step in recombining two individuals is creating exact copies of the parent individuals. With probability $1 - P_{cross}$ these individuals are returned as offspring and no recombination is performed. Otherwise, crossing over is performed on either the allocation, the task binding or the priority assignment of flows.

With probability $P_{cross-alloc}$, a one-point crossover operation is applied

to the allocation vectors A_1 and A_2 of the parents: First we randomly define the position j where to perform the crossover, then we create the allocation vector A_{new1} for the first offspring as follows.

$$A_{new1}[i] = A_1[i], \text{ if } 1 \leq i < j$$
$$A_{new1}[i] = A_2[i], \text{ if } j \leq i \leq |R|$$

Similarly, A_{new2} is created. After this exchange in the allocation of resources, the repair method is called, to ensure, that for all tasks there is at least one resource allocated, on which the task can be performed.

If the crossover is not done within the allocation vector, it is performed with probability $P_{cross-bind}$ within the binding of tasks to resources. In detail, a scenario $b \in B$ is randomly determined, for which the crossover of the binding vectors should happen. Then, a one point crossover for the binding vectors $Z_1[b]$ and $Z_2[b]$ of the parents according to the following procedure is performed, where j is a random value in the interval $[1, |T|]$.

$$Z_{new1}[b][i] = Z_1[b][i], \text{ if } 1 \leq i < j$$
$$Z_{new1}[b][i] = Z_2[b][i], \text{ if } j \leq i \leq |T|$$

The binding Z_{new2} can be determined accordingly.

Finally, if the crossover is neither in the allocation nor in the binding of tasks to resources, the crossover happens in the priority vector. For a randomly selected scenario b, the priority vectors $P_1[b]$ and $P_2[b]$ are crossed in one point to produce new priority vectors P_{new1} and P_{new2} following a similar procedure as described above.

Mutation First, an exact copy of the individual to be mutated is created. With probability P_{mut} no mutation takes place and the copy is returned. Otherwise, the copy is modified with respect to either the allocation, the task binding, or the priority assignment.

We mutate the allocation vector with probability $P_{mut-alloc}$. To this end, we randomly select a resource i and set $A_{new}[i] = 0$ with probability $P_{mut-alloc-zero}$, otherwise we set $A_{new}[i] = 1$. After this change in the allocation vector, the repair method is called, which changes infeasible bindings such that they all map tasks to allocated resources only.

In case the mutation does not affect the allocation vector, with probability $P_{mut-bind}$ we mutate the binding vector $Z_{new}[b]$ for a randomly determined scenario $b \in B$. That is we randomly select a task and map it to a resource randomly selected from the specification. If the resource is not yet allocated in this solution, we additionally allocate it.

If we do neither mutate the allocation nor the binding, we mutate the priority vector for a randomly selected scenario b. We just exchange two flows within the priority list $P_{new}[b]$.

3. Software Architecture

In the following the software architecture as shown in Figure 2 is discussed focusing on the interface between the benchmark application and the multiobjective optimizer. The two main questions are: (i) which elements of the optimization process should be part of the implementation of the benchmark application and (ii) how to establish the communication between the two parts?

3.1. *General Considerations*

Essentially, most proposed multiobjective optimizers differ only in their selection operators: how promising individuals are selected for variation and how it is decided which individuals are removed from the population. Accordingly, most studies comparing different optimizers keep the representation of individuals and the variation operators fixed in order to assess the performance of the selection operators, which form the problem independent part of the optimization process [7,6].

Consistently with this approach, the packet processor benchmark module consists of the individual handling including their representation and the objective function evaluation, as well as the variation operators (see Figure 2 and its extension in Figure 5).

The division of the optimization process into two parts raises the problem of communication between the benchmark application and the optimizer. Several options can be considered: restricting oneself to a specific programming language that is available on many platforms, e.g. C or Java, would allow to provide the modules as library functions. However, coupling two modules which are written in different programming languages would then be difficult and it would in any case be necessary to re-compile or at least re-link the program for each optimizer. Alternatively, special communication mechanisms like UNIX sockets which are independent of the programming language could be used. The drawback is, though, that these mechanisms are not supported on all platforms.

We have therefore decided to implement the benchmark and the optimizer as separate programs which communicate through text files. The use of text files guarantees that any optimizer can be coupled to the benchmark

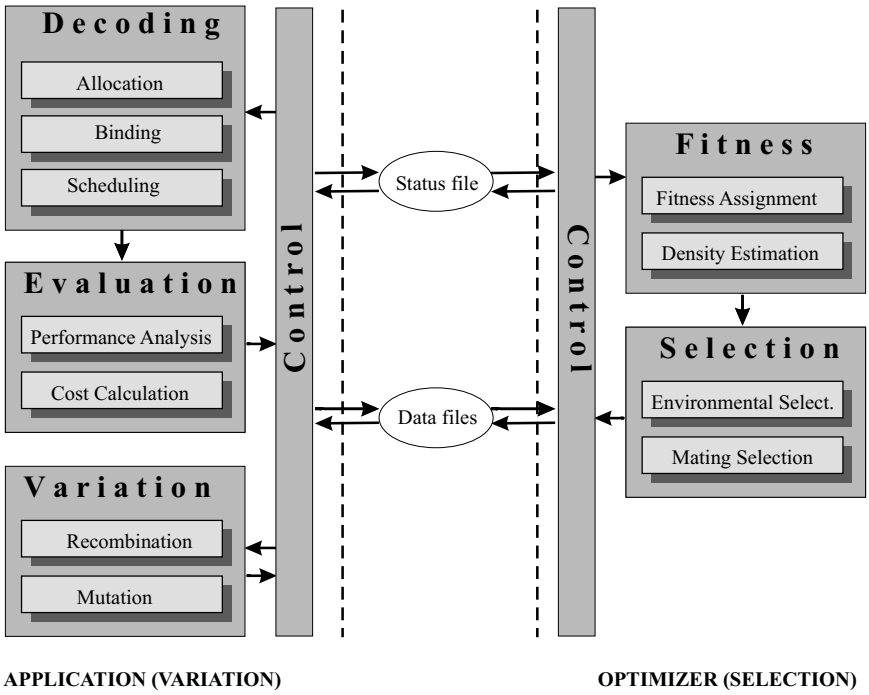

Fig. 5. Overview of the separation between benchmark and optimizer.

even if the two programs were written in different programming languages and run on different machines with different operating systems as long as both have access to a common file system. This flexibility does certainly not come for free. The additional overhead in running time, however, is minimal and can be neglected in the case of real world applications as a series of tests have shown [2].

The interface developed for this benchmark application has been proposed and described in detail elsewhere [2]. It is applicable in a much wider range of scenarios since the chosen separation between selection and variation is suitable for most evolutionary multiobjective optimizers as well as for many other stochastic search algorithms. Additionally, the interface definition provides means for extensions to adjust it to specific needs. In the following we describe the main characteristics of the interface and especially its communication protocol.

3.2. *Interface Description*

As mentioned above the two parts shown in Figure 5 are implemented as separate programs. Since the two programs run as independent processes, they need a method for synchronization. The procedure is based on a hand shake protocol which can be described using two state machines (see Figure 6). In general only one process is active at one time. When it reaches a new state it writes this state, encoded as a number, to a text file. During that time the other process has been polling this state file and now becomes active, while the first process starts polling the state file. Specifically it works as follows.

Both programs independently perform some initialization, e.g. reading of parameter files. During this step the benchmark application generates the initial population and evaluates the individuals. It then writes the IDs of all individuals and their objective values to a file and changes the state number. As soon as the optimizer is done with its initialization it starts polling the state file ('wait' state in Figure 6). When the state number is changed by the benchmark application, the optimizer reads the data file and selects promising parents for variation (in 'select'). Their IDs are written to another text file. The optimizer can maintain a pool of IDs it might consider again for selection in future iterations. The list of these archived IDs is also written to a text file. Then the state number is changed back. The benchmark program which had been polling the state file now (in 'variate') reads the list of archived IDs and deletes all individuals which are not on

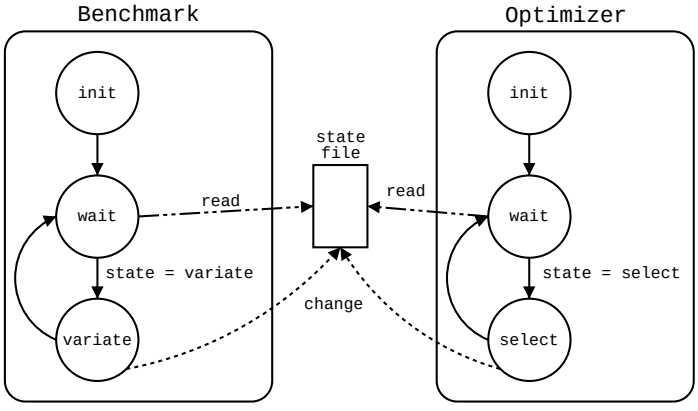

Fig. 6. Handshake protocol: The two processes can be modeled as finite state machines using the state file to synchronize. The data files are not shown.

this list. Then the benchmark reads the IDs of the parents and produces offspring by variation of the respective individuals. They are evaluated, IDs and objective values are written to the text file and the cycle can start again.

Since the optimizer only operates in the objective space it is not necessary to communicate the actual representation of the individuals to the optimizer. The amount of data exchanged is thus small. For the exact specification of the protocol and the format of the text files see Bleuler et al. [2].

4. Test Cases

The packet processor benchmark application has been implemented in Java. The corresponding tool EXPO , which has been tested under Solaris, Linux and Microsoft Windows, provides a graphical user interface that allows to control the program execution (c.f. Figure 7): the optimization run can be halted, the current population can be plotted and individual processor designs can be inspected graphically.

In the following, we will present three problem instances for the packet processor application and demonstrate how to compare different evolutionary multiobjective optimizers on these instances.

4.1. *Problem Instances*

We consider three problem instances that are based on the packet processing application depicted in Figure 3. The set of available resources is the same for all the problem instances. The three problem instances differ in the number of objectives. We have defined a problem with 2 objectives, one with 3 and a scenario including 4 objectives. For all the different instances one objective is in common, the total cost of the allocated resources. The remaining objectives in a problem instance are the performance Ψ_b of the solution packet processor under a given load scenario $b \in B$. In Table 1 the different load characteristics for the remaining objectives are shown. Overall, three different loads can be identified, in *Load 1* all flows have to be processed. In *Load 2* there are only three flows present: real-time voice receive and send, and non-real-time (NRT) packet forwarding. In *Load 3*, the packet processor has to forward packets of flow 'NRT forward' and encrypt/decrypt packets of flows 'NRT encryption' and 'NRT decryption', respectively.

The size of the search space for the given instance can be computed

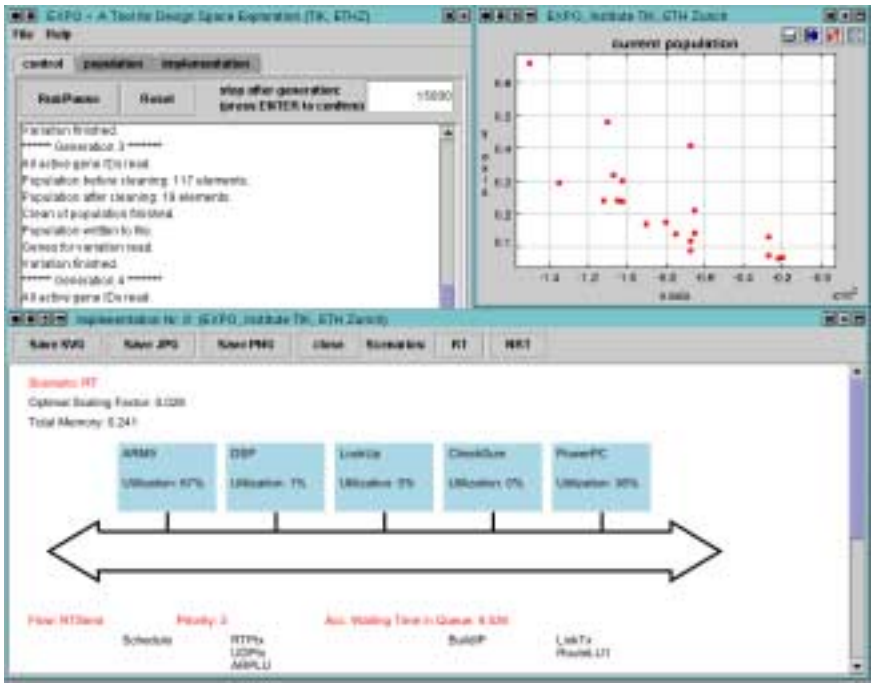

Fig. 7. The user interface of the benchmark application: The main control window in the upper left, a plot the current population in the upper right and a graphical representation of a network processor configuration in the lower part.

Table 1. Loads for the different scenarios for which the architecture should be optimized.

Load scenario		RT send	RT receive	NRT encrypt	NRT decrypt	NRT forward
2 Objectives	Load 1	√	√	√	√	√
3 Objectives	Load 2	√	√	-	-	√
	Load 3	-	-	√	√	√
4 Objectives	Load 1	√	√	√	√	√
	Load 2	√	√	-	-	√
	Load 3	-	-	√	√	√

as follows. In the problem setting, there are 4 resource types on which all the tasks can be performed. Therefore, we have more than 4^{25} possibilities to map the tasks on the resources. Furthermore, the solution contains a priority assignment to the different flows. There are 5! possibilities to assign priorities to the flows. So, if we take into account that there are other

specialized resources available, the size of the search space is $S > 4^{25} \times 5! > 10^{17}$ already for the problem instance with 2 objectives and even larger for the instances with 3 or 4 objectives.

As an example, an approximated Pareto front for the 3-objective instance is shown in Figure 8—the front has been generated by the optimizer SPEA2 [25]. The x-axis shows the objective value corresponding to Ψ_{Load2} under *Load 2* (as defined in Table 1), the y-axis shows the objective value corresponding to Ψ_{Load3}, whereas the z-axis shows the normalized total cost of the allocated resources.

The two example architectures shown in Figure 8 differ only in the allocation of the resource 'Cipher', which is a specialized hardware for encryption and decryption of packets. The performance of the two architectures for the load scenario with real-time flows to be processed is more or less same. However, the architecture with a cipher unit performs around 30 times better for the encryption/decryption scenario, at increased cost for the cipher unit. So, a designer of a packet processor that should have the capability of encryption/decryption would go for the solution with a cipher unit (solution on the left in Figure 8), whereas a designer with no need for encryption would decide for the cheaper solution on the right.

4.2. *Simulation Results*

To evaluate the difficulty of the proposed benchmark application, we compared the performance of four evolutionary multiobjective optimizers, namely SPEA2 [25], NSGA-II [8], SEMO [15] and FEMO [15], on the three aforementioned problem instances.

For each algorithm, 10 runs were performed using the parameter settings listed in Tables 2 and 3; these parameters were determined based on extensive, preliminary simulations. Furthermore, all objective functions were scaled such that the corresponding values lie within the interval $[0, 1]$. Note that all objectives are to be minimized, i.e., the performance values are reversed (smaller values correspond to better performance). The different runs were carried out on a Sun Ultra 60. A single run for 3 objectives, a population size of 150 individuals in conjunction with SPEA2 takes about 20 minutes to complete.

In the following we have used two binary performance measures for the comparison of the EMO techniques: (1) the additive ε-quality measure [28] and (2) the coverage measure [27]. The ε-quality measure $I_{\varepsilon^+}(A, B)$ returns the maximum value d, which can be subtracted from all objective values

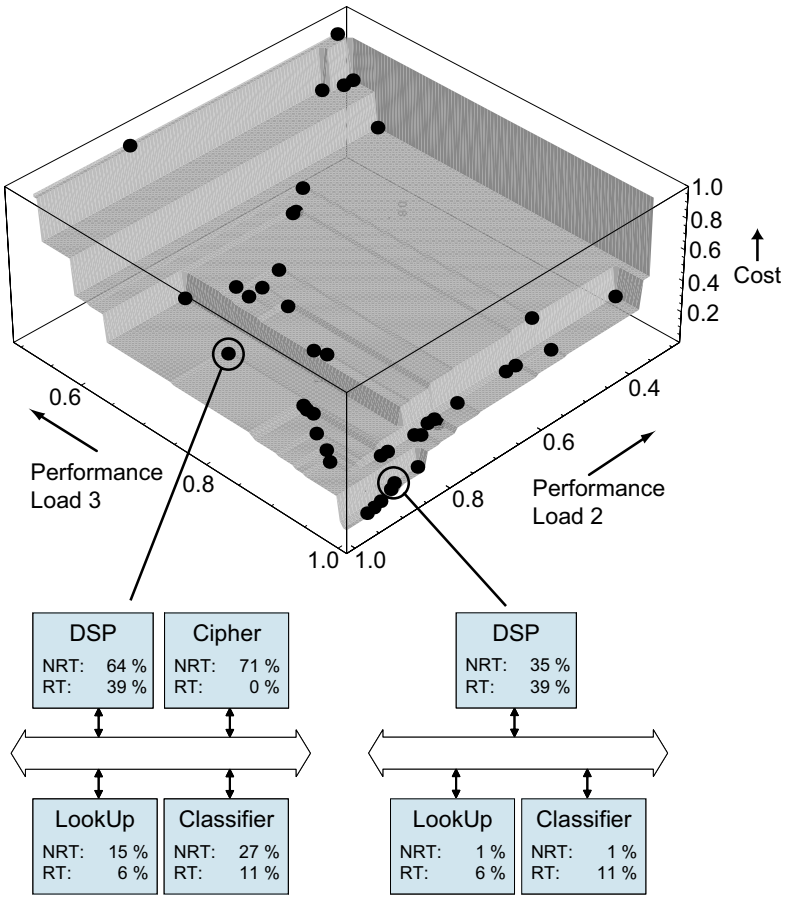

Fig. 8. Two solution packet processor architectures annotated with loads on resources for the different loads specified in Table 1.

Table 2. Parameters for population size and duration of runs dependent on the number of objectives.

# of objectives	population size	# of generations
2	100	200
3	150	300
4	200	400

for all points in the set of solutions A, such that the solutions in the shifted set A' equal or dominate any solution in set B. If the value is negative the

Table 3. Probabilities for mutation and crossover (cf. Section 2)

Mutation		P_{mut}	=	0.8
→	Allocation	$P_{mut-alloc}$	=	0.3
		$P_{mut-alloc-zero}$	=	0.5
→	Binding	$P_{mut-bind}$	=	0.5
Crossover		P_{cross}	=	0.5
→	Allocation	$P_{cross-alloc}$	=	0.3
→	Binding	$P_{cross-bind}$	=	0.5

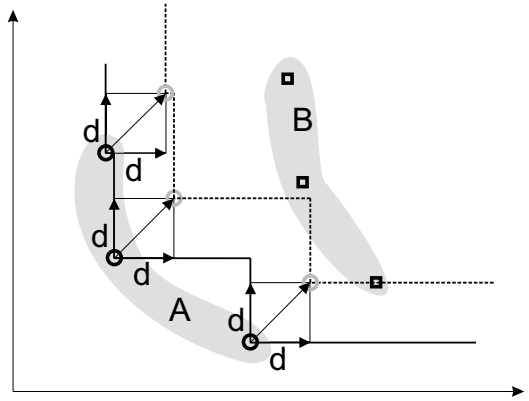

Fig. 9. Illustration of the additive ε-quality measure $I_{\varepsilon+}$, here $I_{\varepsilon+}(A, B) = d$ where $d < 0$ as A entirely dominates B.

solution set A entirely dominates the solution set B. Formally, this measure can be stated as follows:

$$I_{\varepsilon+}(A, B) = \max_{b \in B} \left\{ \min_{a \in A} \left\{ \max_{0 \leq i \leq dim} \{a_i - b_i\} \right\} \right\}$$

Figure 9 shows a graphical interpretation of the additive ε-quality measure. The coverage measure $C(A, B)$, which is used as an additional reference here, returns the percentage of solutions in B which are dominated by or equal to at least one solution in A.

Due to space restrictions, not all results can be presented here. Instead, we will focus on the main results. Overall, we can state that there exist differences in the performance of different evolutionary algorithms on the packet processor design space exploration benchmark problem. From the results of the benchmark study, we can see that the SPEA2 and NSGA-II perform comparably well. For the problem with 2 objectives, NSGA-II

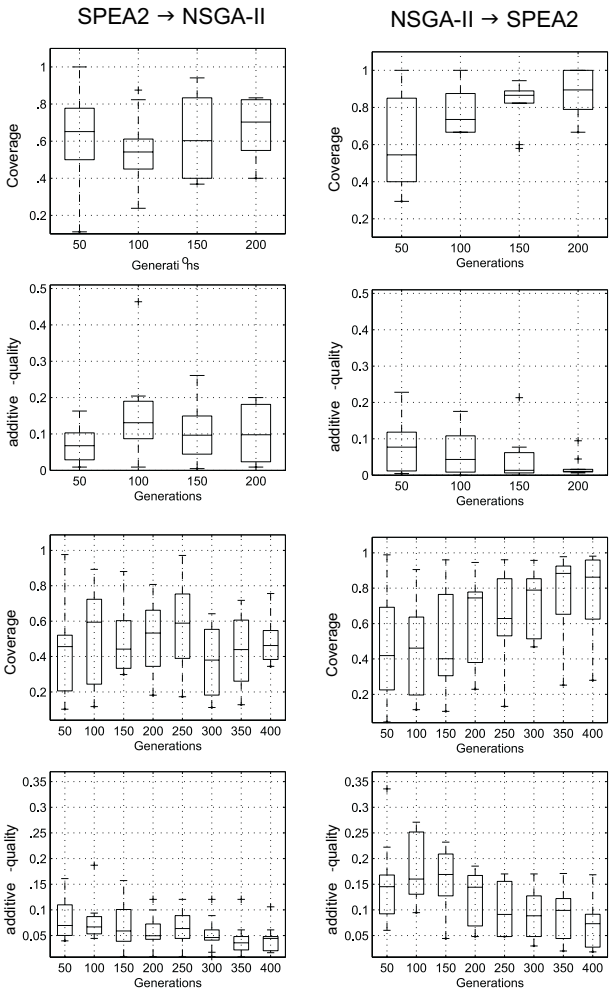

Fig. 10. Comparison of SPEA2 and NSGA-II for 2 and 4 objectives

performs slightly better than SPEA2, whereas SPEA2 performs better for 4 objectives; this confirms results presented in a study by Zitzler, Laumanns, and Thiele [25]. The distributions of the coverage and the additive ε-quality measure values between SPEA2 and NSGA-II are depicted in Figure 10. In both the cases, for 2 and 4 objectives, NSGA-II achieves a better value for coverage over SPEA2, but for more objectives, SPEA2 finds solutions that lead to smaller values for $I_{\varepsilon+}$ than NSGA-II.

Furthermore, we can see that FEMO, a simple evolutionary optimizer with a fair selection strategy, performs worse than the other algorithms for all the test cases (see Figure 11 for details). Note that because of the implementation of selection in FEMO, it is possible that a solution is selected for reproduction multiple times in sequence. This behavior can decrease the impact of recombination in the search process with FEMO. With respect to both coverage and especially the additive ε-quality measure, SPEA2 is superior to FEMO. SEMO, in contrast, performs similarly to FEMO for the case with 2 objectives; however, SEMO shows improved performance with increasing number of objectives.

5. Summary

This chapter presented EXPO, a computer engineering application that addresses the design space exploration of packet processor architectures. The underlying optimization problem is complex and involves allocating resources, binding tasks to resources, and determining schedules for the usage scenario under consideration. The goal is to minimize the cost of the allocated resources and to maximize the estimated performance of the corresponding packet processor architecture for each distinct usage scenario. Especially the last aspect, performance estimation, is highly involved and makes the use of black-box optimization methods necessary. As shown in the previous section, the application reveals performance differences between four selected multiobjective optimizers with several problem instances. This suggests that EXPO is well suited as a multiobjective benchmark application.

Moreover, the EXPO implementation provides a text-based interface that follows the PISA specification [2]. PISA stands for 'platform and programming language independent interface for search algorithms' and allows to implement application-specific parts (representation, variation, objective function calculation) separately from the actual search strategy (fitness assignment, selection). Therefore, EXPO can be downloaded as a ready-to-use package, i.e., pre-compiled for different platforms; no modifications are necessary to combine it with arbitrary search algorithms. In addition, several multiobjective evolutionary algorithms including SPEA2 [25] and NSGA-II [8] as well as other well-known benchmark problems such as the knapsack problem and a set of continuous test functions [24,9] are available for download at the PISA website *http://www.tik.ee.ethz.ch/pisa/*. All PISA compliant components, benchmarks and search algorithms, can be arbitrarily com-

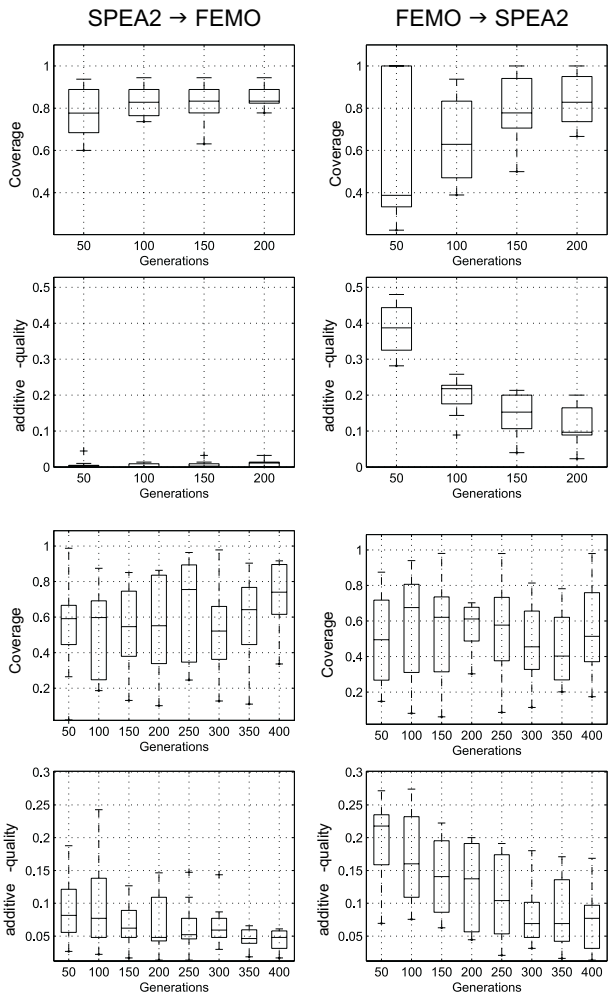

Fig. 11. Comparison of SPEA2 and FEMO for 2 and 4 objectives

bined without further implementation effort, and therefore this interface may be attractive for other researchers who would like to provide their algorithms and applications to the community.

Acknowledgments

This work has been supported by the Swiss Innovation Promotion Agency (KTI/CTI) under project number KTI 5500.2 and the SEP program at

ETH Zürich under the poly project TH-8/02-2.

References

1. A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.
2. S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA — a platform and programming language independent interface for search algorithms. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494–508, Berlin, 2003. Springer.
3. T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. *Journal on Design Automation for Embedded Systems*, 3(8):23–58, 1998.
4. S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2003.
5. S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, 41(5):641–665, April 2003.
6. C. A. Coello Coello, D. A. Van Veldhuizen, and G. B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, New York, 2002.
7. K. Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley, Chichester, UK, 2001.
8. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA–II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.
9. K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. Scalable multi-objective optimization test problems. In *Congress on Evolutionary Computation (CEC)*, pages 825–830. IEEE Press, 2002.
10. R. P. Dick and N. K. Jha. MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-synthesis of Hierarchical Heterogeneous Distributed Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, 1998.
11. M. Franklin and T. Wolf. A network processor performance and design model with benchmark parameterization. In P. Crowley, M. Franklin, H. Hadimioglu, and P. Onufryk, editors, *Network Processor Design: Issues and Practices, Volume 1*, chapter 6, pages 117–140. Morgan Kaufmann Publishers, 2003.
12. D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Englewood Cliffs, N.J., 1994.
13. A. Jaszkiewicz. Do multiple-objective metaheuristics deliver on their

promises? a computational experiment on the set-covering problem. *IEEE Transactions on Evolutionary Computation*, 7(2):133–143, 2003.
14. J. Knowles and D. Corne. Instance generators and test suites for the multiobjective quadratic assignment problem. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 295–310, Berlin, 2003. Springer.
15. M. Laumanns, L. Thiele, and E. Zitzler. Running time analysis of multiobjective evolutionary algorithms on pseudo-boolean functions. *IEEE Transactions on Evolutionary Computation*, 2004. Accepted for publication.
16. M. Naedele, L. Thiele, and M. Eisenring. Characterising variable task releases and processor capacities. In *14th IFAC World Congress 1999*, pages 251–256, Beijing, July 1999.
17. A. Pimentel, P. Lieverse, P. van der Wolf, L. Hertzberger, and E. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.
18. L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *First Workshop on Network Processors at the 8th International Symposium on High-Performance Computer Architecture (HPCA8)*, pages 30–41, Cambridge MA, USA, February 2002.
19. L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proc. 39th Design Automation Conference (DAC)*, pages 880–885, New Orleans, LA, June 2002. ACM Press.
20. L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *Network Processor Design: Issues and Practices*, volume 1, chapter 4, pages 55–90. Morgan Kaufmann Publishers, 2003.
21. L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors – models and algorithms. In *Proc. 1st Workshop on Embedded Software (EMSOFT)*, Lecture Notes in Computer Science 2211, pages 416–434, Lake Tahoe, CA, USA, 2001. Springer Verlag.
22. L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.
23. D. Thierens. Convergence time analysis for the multi-objective counting ones problem. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 355–364, Berlin, 2003. Springer.
24. E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195, 2000.
25. E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In K. Giannakoglou et al., editors, *Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001)*, pages 95–100. International Center for Numerical Methods in Engineering

(CIMNE), 2002.
26. E. Zitzler, J. Teich, and S. S. Bhattacharyya. Multidimensional exploration of software implementations for DSP algorithms. *Journal of VLSI Signal Processing*, 24(1):83–98, February 2000.
27. E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.
28. E. Zitzler, L. Thiele, M. Laumanns, C. M. Foneseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.

INDEX