# Iteration-based Trade-off Analysis of Resource-aware SDF*

Yang Yang[1], Marc Geilen[1], Twan Basten[1,2], Sander Stuijk[1], Henk Corporaal[1]
[1]Department of Electrical Engineering, Eindhoven University of Technology, Netherlands
[2]Embedded Systems Institute, Eindhoven, Netherlands
{y.yang, m.c.w.geilen, a.a.basten, s.stuijk, h.corporaal}@tue.nl

***Abstract*—Synchronous dataflow graphs (SDFGs) are widely used to model streaming applications such as signal processing and multimedia applications in embedded systems. Trade-off analysis between performance and resource usage of SDFGs allows designers to explore implementation alternatives of a system while meeting its performance requirements and resource constraints. This type of analysis is computationally very challenging, particularly when resources may be shared among computations. With resource sharing, system scheduling decisions lead to a combinatorial explosion in the number of scheduling alternatives to be explored. We present a new approach to explore the trade-offs in a such systems. It breaks analysis down in iterations of dataflow graph execution and uses a max-plus algebra semantics. The experimental results on a set of realistic benchmark models show that the new iteration-based approach and the traditional time-based analysis approach complement each other. None of the two approaches dominates the other in terms of quality of the analysis results and analysis time. The two approaches combined give the highest quality result.**

***Index Terms*—Synchronous dataflow, Max-plus Algebra, Design-space Exploration**

## I. Introduction

Embedded systems can be found almost everywhere, in smart phones, e-book readers, portable media players and digital printers. An important class of applications, widely found in those electronic devices, are streaming applications like image, audio and video processing. Since those embedded systems are always resource constrained, resources have to be shared among many tasks. At the same time, it is becoming practice in system design to use a reference platform that will be carefully tailored to the specific set of applications considered, in order to reduce time-to-market and development costs. Designers have to carefully tune the platform parameters (number of cores, size of buffers, etc.) to meet performance requirements. Model-based design methods promise to solve such design challenges, and deliver a shorter development cycle while ensuring functional correctness and guaranteed performance of products.

For finding a good match between application and architecture or platform instance, the Y-chart methodology is often employed in embedded system design [2], [12]. According to this method, designers first specify application and architecture aspects of an embedded system separately and then map them together to do performance analysis. From the analysis, designers can tune platform parameters and resource mapping to satisfy application requirements and exploit available trade-offs.

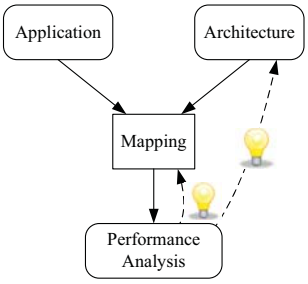
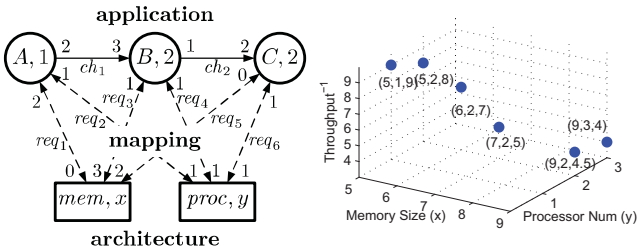



Fig. 1. Y-Chart Methodology [2], [12]

The SDFG model and its variants are often applied to model streaming applications as well as platforms and their mapping. However, resources are implicitly modeled in SDFGs. For example, a buffer between two application tasks is modeled as tokens in a feedback dependency edge between those tasks and processors can be modeled as dependencies between task invocations. When not explicitly modeled, resources are assumed unlimited. In this model resources are always private or resource access is statically ordered. Resource-Aware SDFG (RASDFG) [21] is a particular extension of the SDFG [13] model based on the Y-chart methodology to separate and model resources explicitly. It combines *application*, *architecture* and *mapping* aspects of a system into one graph. Fig. 2(a) shows an example of RASDFG. Moreover, it allows dynamic sharing of resources. The design space exploration addressed in this paper targets *dimensioning* of a given architecture, i.e., what should be the capacities of different resources in the architecture to allow concurrent execution of tasks, and investigates the trade-off with performance (throughput). RASDFG is used to analyze these trade-offs between throughput and resource usage and to dimension resource, given an architecture and a set of fixed types of resources [21], [22].

**Contributions.** An iteration-based approach is proposed to explore trade-offs in an RASDF graph. Earlier work on analyzing RASDFGs [21], [22] does not exploit the fact that system execution occurs in iterations. In this paper, we develop a novel iteration-based trade-off analysis technique

*This work has been carried out as part of the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.




(a) An RASDFG (b) Design space

Fig. 2. An illustrative example

for RASDFG, aiming for an improved quality of the results and efficiency of the analysis. The approach is grounded in max-plus algebra [4], [11], which provides a natural means to capture iteration-based execution of dataflow applications. It turns out that the newly developed method improves quality and/or efficiency of the analysis in some, but not all cases. Best quality results can be obtained by a combination of the new iteration-based method with the traditional method.

## II. Preliminaries

To illustrate our approach, we introduce an example that is used throughout this paper. Assume we have a streaming application that consists of 3 tasks ($A$, $B$ and $C$). We map it to a reference platform with multiple processors and a shared memory for which neither the size of the memory ($x$) nor the number of cores ($y$) are decided. Fig. 2(a) shows its system model as an RASDFG. The trade-offs in its design space are shown in Fig. 2(b).

The RASDFG in Fig. 2(a) includes three parts of the given system:

**application.** The circular nodes, *actors*, represent computations. Actor names and execution times are shown inside the nodes. Actors communicate through FIFO *channels* (solid directed edges) using data items called *tokens*.

**architecture.** The rectangular nodes represent (shared) *resources* that are needed by the computations. Resource names and amounts are shown inside the nodes.

**mapping.** Claims and releases of resources by actors are denoted by *request edges* (the bidirectional dashed edges).

An essential property of an RASDFG is that every time an actor fires (executes), it consumes/claims the same amount of tokens/resources from its input ports/resource providers and produces/releases the same amount of tokens/resources to its output ports/resource providers. These amounts are called the *rates*, and are attached to both ends of the edges in the figure. For example, in Fig. 2(a), actor $A$'s execution time is 1 time unit and it claims 2 units of memory when it starts firing, and outputs 2 tokens on channel $ch_1$ when it ends firing. It does, however, not yet release the memory. Actor $B$ consumes 3 tokens from channel $ch_1$ and claims 1 unit of memory when it starts firing, and outputs 1 token on $ch_2$ and releases 3 units of memory that were claimed by actor $A$ and itself, when it ends firing.

RASDFGs are strictly more expressive than regular Synchronous Dataflow graphs. If every resource is claimed by only one actor and released by only one actor, then an RASDFG can be converted to an SDFG. However, if resources are claimed and/or released by multiple actors, such transformation does in general not exist. Analyzing performance-resource usage trade-offs for RASDFGs is a fundamentally more difficult problem than similar analyses on regular SDFGs, like the throughput-buffer sizing trade-off analysis of [15]. In an RASDFG execution, actors may compete for resources, and the order of firings impacts resource usage and performance. As a result, different orderings need to be investigated and the size of the execution state space grows rapidly for more complex RASDFGs and therefore an exhaustive exploration is not practically feasible for those graphs. State-space reduction techniques are needed for efficient design space exploration.

One important property of SDFGs is preserved by RASDFGs, namely the fact that execution occurs in *iterations*. An iteration is the minimal non-empty set of actor firings that does not have a net effect on the system state (available data tokens and resources) of an RASDFG, when executed. For the example of Fig. 2, an iteration consists of three firings of $A$, two of $B$, and one of $C$. Therefore, an execution of the graph can be conveniently partitioned into separate, but pipelined, iterations.

The rest of this paper is structured as follows. The next section discusses related work. Sec. IV introduces RASDFGs and their operational semantics in the iteration-based approach. Sec. V discusses state-space pruning techniques based on max-plus algebra. An experimental evaluation is given in Sec. VI. Sec. VII concludes.

## III. Related Work

Much work has been done on analyzing SDFG throughput and on synthesizing schedules that minimize the resource of buffer sizes [3], [8], [10], [14], [18], [23]. Only recently, trade-off analysis for SDFGs [17], [21], [22], [24] is investigated. [17] uses bottleneck analysis to explore distributed buffer size configurations efficiently while [22] applies this approach to the more general RASDFG model. [23], [24] investigate trade-offs between cost and performance by differently partitioning actors to software and hardware implementations, assuming software and hardware realisations have different cost and performance. They use constraint programming to find a schedule that satisfies a given throughput constraint and has the minimal total size of distributed buffers. Then Pareto optimization is applied to the discovered solutions characterized by their software, hardware and buffer costs. They investigate the binding problem rather than the resource dimensioning problem that is investigated in this paper. Existing trade-off analysis work assumes traditional time-based representations of system execution. Except for the work on RASDFGs [21], [22], all this work also limits exploration to specific execution policies, such as self-timed execution or periodic execution. Such policies are no longer optimal when resource sharing is allowed. Recently, [7], [9] introduce an iteration-based throughput analysis for an SDFG variant called scenario-aware SDFGs. Scenario-aware dataflow allows dynamic actor execution time changes between iterations. The analysis is

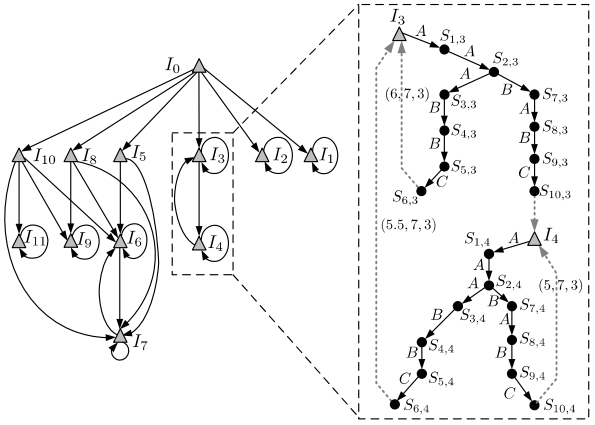



Fig. 3. State space example in max-plus view

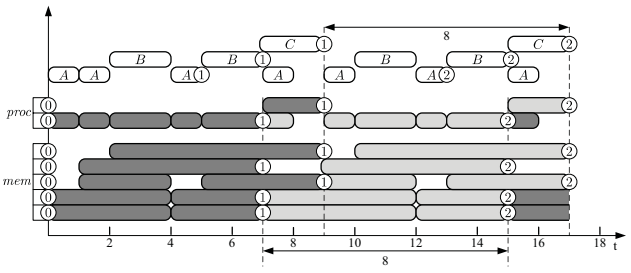



Fig. 4. An execution chart of our example

based on a max-plus representation of SDFG execution. Max-plus algebra is also used to analyze the performance of Petri-nets [5], [6], which is a more general model of computation encompassing SDFGs. Inspired by this work, we apply max-plus algebra for RASDFG analysis, to compute throughput-resource usage trade-offs in the design space of the RASDFG. The approaches of [7], [9] are not directly applicable because of non-determinism in RASDFG execution. The approaches of [5], [6] do not fully exploit the characteristics of RASDFGs and are not directly applicable to trade-off analysis.

## IV. Max-plus View on RASDFGs

### A. Motivation of the new approach

In order to explore the trade-offs of a given RASDFG, we need to explore its different executions. As mentioned, RASDFGs execute in iterations. Like a regular SDFG, also an RASDFG requires *consistency* and has a *repetition vector* [13], [21]. An RASDFG is consistent if and only if there exists a non-trivial repetition vector $q$, which assigns a non-zero number of firings to every actor, such that, after any sequence of actor firings conforming to $q$, an *iteration*, the number of tokens in the channels as well as the amount of resources are equal to their initial state values. The repetition vector of the example of Fig. 2 equals $q(A) = 3$, $q(B) = 2$, $q(C) = 1$.

The max-plus view on RASDFG execution uses the production times of tokens and the release times of resources to capture the state of an RASDFG after a sequence of actor firings. By exploring the firing order of actors inside one iteration and checking for recurrence od states after every complete iteration, we explore the trade-offs in a given RASDFG on an iteration-by-iteration basis.

Fig. 3 shows part of the state space of the model of Fig. 2(a) with memory size $x = 9$ and $y = 3$ processors, generated according to an iteration-based approach. Triangular states $I_k$ represent the states in the iteration state space reached after a number of whole iterations and edges with arrows denote single iterations. The right parts of the figure zooms into the iterations between $I_3$ and $I_4$ and shows the intra-iteration states $S_{i,j}$ ($i$th state in $j$th iteration explored) and individual actor firings (arrows labeled with actor names). Dashed back edges denote occurrences of recurrent states in the state space (for instance, $S_{6,3}$ is identical to $I_3$) and the trade-offs (iteration period, memory and number of processors) of the corresponding cycles are annotated with them. Iteration period is the (average) time taken for one iteration and is thus inversely proportional to the throughput.

Since the amounts of tokens and resources do not change after any number of complete iterations, to detect recurrent states, we only need to compare the production times of tokens and resources, not their quantities (precise definitions are given in the following section). We therefore also only need to store the triangle states of every iteration. For example, instead of 20 states (the states in the dashed box), we only have to store 2 states ($I_3$ and $I_4$) in Fig. 3. This property sharply reduces checks for recurrent states and the size of the state space stored compared to the traditional state-space exploration approach, which we refer to as the *time-based* approach. For the example, only 123 iteration states (triangles) need to be stored for a full exploration when using the iteration-based exploration, in comparison to 220 states when using the time-based exploration of [22], to explore the state space to a depth of just 4 iterations.

### B. Iteration-based State and Execution of RASDFG

To formalize our iteration-based approach, we use max-plus algebra to capture the execution of a given RASDFG. Fig. 4 shows one of its executions with memory size $x = 5$ and $y = 2$ processors, where the horizontal axis is time and the vertical axis shows resources. We separate different resources ($mem$, $proc$) in the vertical axis of the chart, into their individual units. At the top, it shows a Gantt chart with the individual actor firings. The chart shows the acquisition and release of resources. The small circles with enclosed numbers denote the end of each iteration of the graph and indicates when that resource unit is ultimately released for the execution of the iteration; the number inside the circle is the iteration count. After the second iteration, in lighter grey color, we observe that the resource release times are identical to the release times after the first iteration, except that they are all shifted forward by 8 time units. Thus, the execution may go into a periodic phase, repeating this behavior forever with period 8.

In the time-based state-space exploration, a state at time $t$ keeps information about active actors of an RASDFG at $t$ and the token distribution in channels and resources. This representation is intuitive but it is impossible to distinguish iteration states, since the firings of different actors reach the end of an iteration at different times; the iterations overlap

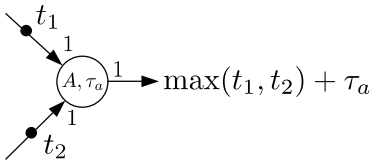



Fig. 5. Max-plus semantics for actor firing

in time, are pipelined. For example, actor $A$ finishes the 1st iteration at time 5 while actor $B$ finishes at time 7. Actor $C$ finishes at time 9, while actor $A$ has already started the next iteration at that moment in time.

In order to represent the iteration state of an RASDFG, we define the *time stamp* of each token as the time it was produced, i.e., written into the channel, for a data token, and the time it was released, for resource tokens. We use this representation for a nested exploration strategy that explores the scheduling possibilities inside a single iteration and then only constructs a state-space of iteration states in memory, labeled with the resource usage and time taken by the iteration. Hence, the new technique is based on maintaining time stamps of tokens indicating their first moment of availability. This time stamp evolution of tokens is illustrated with Fig. 5. When actor $A$ fires with execution time $\tau_a$, it consumes two tokens, with time stamps $t_1$ and $t_2$. Assuming a self-timed execution, the actor starts as soon as both tokens are available, i.e., at time $\max(t_1, t_2)$ and thus it completes and produces a new token with time stamp $\max(t_1, t_2)+\tau_a$. Hence, the process of actors firing and the evolution of time stamp values of tokens can be captured by max-plus algebra equations [1], [11].

We now briefly introduce some notation for max-plus algebra, only as far as it is used in the following sections. Following max-plus algebra notation, we use $\varepsilon$ to denote $-\infty$, important in the algebra as the neutral element of the $\max$ operator, for $a \in \mathbb{R} \cup \{\varepsilon\} = \mathbb{R}_{\max}$, we have $\max(a, \varepsilon) = \max(\varepsilon, a) = a$ and the zero element of the addition operator, $a + \varepsilon = \varepsilon + a = \varepsilon$.

For a vector $\bar{a}$, we use $\|\bar{a}\| = \max\{a_i\}$ to denote its norm and if the norm is larger than $\varepsilon$, $\bar{a}^{norm} = \bar{a} - \|\bar{a}\|$ to denote its normalized vector. When two vectors $\bar{a}$ and $\bar{b}$ have the same length $n$ and for all $i \leq n$ it holds that $a_i \leq b_i$, then we say that $\bar{a}$ dominates $\bar{b}$ and we use $\bar{a} \preceq \bar{b}$ to represent this. For more details, we refer to [1], [11].

A finite execution $\sigma$ is defined as a finite sequence of actor firings with their starting times. For example, the execution in Fig. 4 can be written as $\sigma = (A, 0)(A, 1)(B, 2)(A, 4)\ldots$. We use a counting vector $\gamma(\sigma) = [\gamma_a(\sigma) \mid a \in A]$ with $A$ the set of actors, to denote the total number of firings $\gamma_a(\sigma)$ of each actor $a$ in the execution $\sigma$. The 'state' of an RASDFG can be defined by the locations and time stamps of its data and resource tokens. Although the number and locations of tokens may vary with the firings within an iteration, they return to their original values and places at the end of the iteration. Then only the time-stamps have hanged. We use the notation $\psi_c(\sigma) = \{(m_1, \tau_1), (m_2, \tau_2), \ldots, (m_k, \tau_k)\}$ to denote the time stamps of the tokens in the channel $c$ after $\sigma$, where $(m_i, \tau_i)$ means that there are $m_i$ tokens with the same time stamp $\tau_i$. For a resource $r$, we define $\psi_r(\sigma)$ in a similar way. All *data tokens* are initialized to 0, while *resource tokens* are initialized to $\varepsilon$ (for reasons explained below). The state of an RASDFG after a finite execution $\sigma$ is defined, by the combination of the state of the channels and the state of the resources, as

$$\Psi(\sigma) = \left[ \begin{array}{c|c} \psi_c(\sigma) & c \in C, r \in R \\ \psi_r(\sigma) & \end{array} \right]$$

For example, in Fig. 4, the state of the example RASDFG after a finite execution $\sigma = (A, 0)(A, 1)(B, 2)$ is:

$$\Psi(\sigma) = \left[ \begin{array}{l} \{(1, 2)\}_{ch_1} \\ \{(1, 4)\}_{ch_2} \\ \{(3, 4)\}_{mem} \\ \{(1, \varepsilon), (1, 4)\}_{proc} \end{array} \right]$$

There is one token in channel $ch_1$ with time stamp 2, one in channel $ch_2$ with time stamp 4, 3 memory resource tokens with time stamps 4, one still unused processor with time stamp $\varepsilon$, and one processor resource token with time stamp 4.

With a fixed ordering of the channels and representing individual tokens, we can alternatively represent states in vector form for simplicity. For example, the above state can be written as

$$\zeta(\sigma) = \left[ \begin{array}{ccccccc} 2 & 4 & 4 & 4 & 4 & \varepsilon & 4 \end{array} \right]$$

The first entry in the vector $\zeta(\sigma)$ corresponds to the element $(1, 2)_{ch1}$ from the state $\Psi(\sigma)$. The third, fourth and fifth elements correspond to $(3, 4)_{mem}$. Every entry $(m_i, \tau_i)$ in the state gets expanded into $m_i$ entries in the vector $\zeta(\sigma)$ with value $\tau_i$.

Different resource allocation policies will lead to different states, and the number of possible policies can be very large due to its combinatorial nature. Therefore, we need a strategy to find an optimal policy. Sec. V provides a throughput optimal resource allocation policy to solve this problem.

### C. Throughput and Resource usage computation

Since the time stamps in the state of a given execution keep growing as the execution continues, we normalize the state's time stamps to check for recurrence in the state space by only comparing the relative differences of the time stamps. We therefore store the max-plus normalization of the vector $\zeta(\sigma)$ in memory during state-space exploration. In the following, we use $\sigma_i$ to denote an execution that contains $i$ complete iterations. Assume the execution first visits its recurrent state after the $n_1$th iteration and revisits it after the $n_2$th iteration, then the execution between the $n_1$th and $n_2$th iteration forms a cycle in the state space. Assume that after $k$ iterations of execution, this cycle has been repeated $n$ times, so that $k = n_1 + n(n_2 - n_1)$. The time at which the $i$th iteration completes is $\|\zeta(\sigma_i)\|$ (recall that this is the maximum element in the vector). We can compute its throughput (the average

number of iterations per time unit) with the following equation.

$$\begin{aligned} Thr(\sigma) &= \lim_{k\to\infty} \frac{k}{\|\zeta(\sigma_k)\|} \\ &= \lim_{n\to\infty} \frac{n_1 + n\cdot(n_2 - n_1)}{\|\zeta(\sigma_{n_1})\| + n\cdot(\|\zeta(\sigma_{n_2})\| - \|\zeta(\sigma_{n_1})\|)} \\ &= \frac{n_2 - n_1}{\|\zeta(\sigma_{n_2})\| - \|\zeta(\sigma_{n_1})\|} \end{aligned}$$

In the max-plus view, the state vector contains the time stamps of all available resource tokens. The time stamp of a resource token that was ever used is larger than $\varepsilon$ (hence the initialization to $\varepsilon$). Hence, the number of non-$\varepsilon$ time stamp tokens at a state $S_i$ for resource $r$ is the amount of used resource at the state $S_i$ and denoted by $Ru_r(S_i)$. So, the resource usage of an execution $\sigma$ is $Ru_r(\sigma) = \max\{Ru_r(S_i)\}$, for all $S_i \in \sigma$. $Ru(\sigma)$ denotes the vector $[Ru_r(\sigma) \mid r \in R]$.

Different executions may lead to different cycles and have different throughput and resource usage properties. By exploring the design space of a given RASDFG we can find those cycles, their performance and resource usage trade-offs and the corresponding resource arbitration policies. Note that, in Fig. 3, $S_{10,3} = I_4$; at $S_{10,3}/I_4$ the execution transits from one iteration into the next iteration. The dashed edges denote the detections of recurrent states and the numbers denote the performance value and resource usage. For example, an execution $\sigma$ may reach $I_3$ after its first iteration (denoted as $\sigma_1$). At the end of the $3rd$ iteration of $\sigma$, (after $\sigma_3$), it may reach $S_{6,4}$, which is identical to $I_3$ ($S_{6,4}^{norm} = I_3^{norm}$). The path between $I_3$ and $S_{6,4}$ is a cycle and can be repeated forever. Thus a trade-off point is found. In this point, the average number of time units per iteration is $5.5$, the resource usage is 7 for the memory and 3 for the processors, added as a label $(5.5, 7, 3)$ to the dashed edge.

From a given state, different actor firings may generate different next states. For example, at the state

$$S_{2,3} = \begin{bmatrix} \{(2,8),(2,10)\}_{ch_1} \\ \emptyset_{ch_2} \\ \{(2,\varepsilon),(2,0),(1,9)\}_{mem} \\ \{(1,7),(1,9),(1,10)\}_{proc} \end{bmatrix},$$

there are two enabled actors, $A$ and $B$; we have 2 options: firing $A$ (going from $S_{2,3}$ to $S_{3,3}$) or firing $B$ (going from $S_{2,3}$ to $S_{7,3}$). Note that simultaneous firings can always be replaced by a sequence of single firings. Therefore, we only need to consider interleavings of enabled firings at a state.

Given the state space, we can compute the throughput and resource usage of different executions. For example, the states $I_3$ and $S_{6,4}$ in Fig. 3 are

$$I_3 = \begin{bmatrix} \emptyset_{ch_1} \\ \emptyset_{ch_2} \\ \{(2,\varepsilon),(2,4),(3,6),(2,8)\}_{mem} \\ \{(1,3),(1,6),(1,8)\}_{proc} \end{bmatrix}$$

$$S_{6,4} = \begin{bmatrix} \emptyset_{ch_1} \\ \emptyset_{ch_2} \\ \{(2,\varepsilon),(2,15),(3,17),(2,19)\}_{mem} \\ \{(1,14),(1,17),(1,19)\}_{proc} \end{bmatrix}$$

The normalized time stamp vectors for $I_3$ and $S_{6,4}$ are equal, i.e.,

$$\zeta(\sigma_1)^{norm} = \zeta(\sigma_3)^{norm} = \begin{bmatrix} \varepsilon & \varepsilon & -4 & -4 & -2 & -2 & -2 & 0 & 0 & -5 & -2 & 0 \end{bmatrix}$$

We find a cycle $\sigma_{per}$ consisting of the firings leading from $I_3$ to $S_{6,4}$. The norms are $\|\zeta(\sigma_1)\| = 8$ and $\|\zeta(\sigma_3)\| = 19$. So with $\sigma_{pre}$ being the execution leading from the initial state $I_0$ to $I_3$, the throughput for the infinite execution $\sigma = \sigma_{pre}\sigma_{per}^{\omega}$ is

$$Thr(\sigma) = \lim_{n\to\infty} \frac{1 + n\cdot(3-1)}{8 + n\cdot(19-8)} = \frac{2}{11}.$$

In the execution $\sigma = \sigma_{pre}\sigma_{per}^{\omega}$ (called a simple execution), there are 7 non-$\varepsilon$ time stamp memory tokens, so the usage of the memory resource is 7 and similarly for the processors is 3.

By applying Pareto minimization on the metric points we obtained from the exploration of the state space of an RASDFG, we can find the different trade-off points between throughput and resource usage, i.e, the Pareto points in the metric space. For example, in Fig. 3, we can find 4 different Pareto points: $(8, 5, 3)$, $(7, 6, 3)$, $(5, 7, 3)$, $(4, 9, 3)$. In each point, the first value is the average time per iteration (i.e., the inverse of throughput), the second value is the memory usage, the third is the processor usage.

Given the trade-offs in the metric space, designers can tailor their platform to their wishes. The success of this method depends on efficiently pruning the state space of a given RASDFG. The next section provides theoretical results for efficient pruning of the iteration state space.

## V. State Space Pruning Techniques

In order to explore the design space of a given RASDFG, we need to explore different executions of the RASDFG and compute the throughputs and resource usages of those executions. The executions are different in two aspects: the firing order of actors and the resource allocation policy of resource tokens. By exploiting the properties of the two aspects, we can prune the state space of a given RASDFG efficiently.

### A. Pruning Based on Actor Firing Order

The order of actor firings in an RASDFG can be different in different executions. Under the same resource constraints, two executions may have the same number of firings for each actor, and only differ in the order of actor firings leading to different token time stamps. An execution where all time stamps are larger than or equal to some other execution uses the same resources and is slower and is therefore redundant and can be pruned.

*Proposition 1:* Given two executions $\sigma_1$ and $\sigma_2$ such that the actor firing counts are equal, i.e., $\gamma(\sigma_1) = \gamma(\sigma_2)$. If $\zeta(\sigma_1) \preceq \zeta(\sigma_2)$, then for any execution $\sigma_b = \sigma_2\sigma$, execution $\sigma_a = \sigma_1\sigma$ dominates $\sigma_b$ so that its throughput is equal to or better than the throughput of $\sigma_b$, i.e. $Thr(\sigma_a) \geq Thr(\sigma_b)$ and so is it resource usage $Ru(\sigma_a) \preceq Ru(\sigma_b)$.

*Proof:* Since $\gamma(\sigma_1) = \gamma(\sigma_2)$, the numbers of data and resource tokens are equal between $\sigma_1$ and $\sigma_2$. They only differ

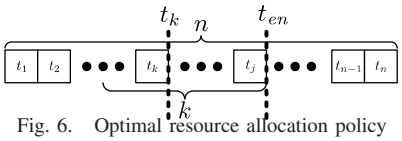


Fig. 6. Optimal resource allocation policy

in time stamp values, which in $\zeta(\sigma_1)$ are no later than in $\zeta(\sigma_2)$. So any actor firing that is possible in $\sigma_b$ after $\sigma_2$ is also enabled in $\sigma_a$ after $\sigma_1$. As $\zeta(\sigma_1) \preceq \zeta(\sigma_2)$, and the max and plus operators are monotonically non-decreasing, for any actor firing sequence $\sigma_s$, the new time stamps vector in $\sigma_a$ will dominate the vector in $\sigma_b$. $\zeta(\sigma_1\sigma_s) \preceq \zeta(\sigma_2\sigma_s)$. So for the $i$th iteration states in $\sigma_a$ and $\sigma_b$ after $\sigma_1$ and $\sigma_2$ respectively, we always have $\zeta((\sigma_a)_i) \preceq \zeta((\sigma_b)_i)$. Hence,

$$Thr(\sigma_a) = \lim_{n\to\infty} \frac{n}{\|\zeta((\sigma_a)_i)\|} \geq \lim_{n\to\infty} \frac{n}{\|\zeta((\sigma_b)_i)\|} = Thr(\sigma_b)$$

Since $\zeta(\sigma_1) \preceq \zeta(\sigma_2)$, $Ru(\sigma_1) \preceq Ru(\sigma_2)$. From $Ru(\sigma_a) = \max(Ru(\sigma_1),\ Ru(\sigma))$ and $Ru(\sigma_b) = \max(Ru(\sigma_2),\ Ru(\sigma))$ it follows that $Ru(\sigma_a) \preceq Ru(\sigma_b)$. ■

From Proposition 1 we know that, if we find that the time stamp vector of a state is dominated by a time stamp vector already existing in the state space with the same firing counts and resource usage, the exploration can backtrack since further exploration cannot lead to a better result.

### B. Pruning Based on Resource Allocation

From Sec. IV we know that there can be many resource allocation policies. However, we want to find throughput-optimal policies to reduce the exploration cost. Based on Proposition 1, we can construct a throughput optimal policy. The policy is illustrated in Fig. 6: an actor is ready to be fired (i.e. enabled) at $t_{en}$ when all data tokens and resource requirements are satisfied. Let $t_{en}$ be the earliest time at which the actor is enabled (there are sufficient data and resource tokens to be able to fire). The $k$ resource tokens it needs are only available at $t_k$. We know that the enable time of the actor $t_{en} \geq t_k$. We always select the $k$ tokens with the *largest* time stamps that are equal to or less than $t_{en}$. This leaves the earlier resource tokens to be used by other actor firings which may thus be able to fire earlier and so improve performance. Since the time stamps are the newest tokens that are available at $t_{en}$, we call the resource token selection policy: ***As New As Possible*** (ANAP) and denote it by $P_{ANAP}$.

*Proposition 2:* The resource allocation policy $P_{ANAP}$ is an optimal policy.

*Proof:* Given an execution at state $s$, an actor with execution time $\tau_a$ is selected for firing at state $s$. Assume the actor enable time is $t_{en}$. Then the output tokens of the actor are time-stamped $\tau_a + t_{en}$ if the firing starts at $t_{en}$. Since $t_{en}$ is the actor enable time, the output time stamps are as small as possible. We only need to compare the remaining time stamps. The channel time stamps are consumed in a FIFO way, so no matter how resource tokens are taken, the remaining channel time stamps are always the same for any resource allocation policy. By using the $P_{ANAP}$ policy, the remaining resource time stamps are the smallest. So, the time vector generated by $P_{ANAP}$ always dominates time vectors generated by another resource allocation policy. From Proposition 1, it then follows that $P_{ANAP}$ is the optimal resource allocation policy. ■

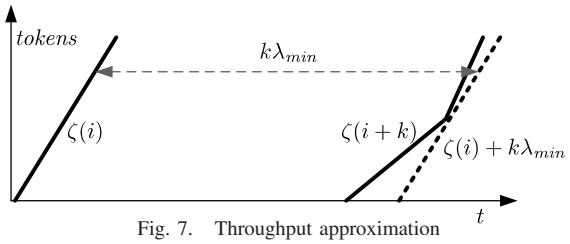


Fig. 7. Throughput approximation

Proposition 2 shows that the $P_{ANAP}$ resource allocation policy provides the best possible throughput. Note that the policy is not necessarily resource optimal. In certain situations, resource could be saved when delaying an actor firing when it is enabled. In our experiments, we perform an exploration with various resource constraints. This allows us to find these trade-off points as well.

### C. Throughput Approximation

In practice, the schedule length is always limited. The state space of an RASDFG without schedule length limit can be very large. To prune the exploration, throughput has to be computed for a bounded number of iterations. However, in some cases, a cycle cannot be detected before reaching the iteration limit. To get useful results out of the aborted explorations, we need to compute an approximation for the throughput. Fig. 7 shows how to compute the approximation.

Assume we have an execution which after the $(i+k)$th iteration reaches a state $\zeta_{i+k}$, which has not been visited before. We can estimate the throughput that is obtainable from $\zeta_i$ by computing the minimal offset $k\lambda_{\min}$ such that $\zeta_{i+k} \preceq \zeta_i + (k\lambda_{\min})$. Then we can always achieve a throughput $Thr = \frac{k}{k\lambda_{\min}} = \frac{1}{\lambda_{\min}}$ by delaying the time stamp vector of $\zeta_{i+k}$ to $\zeta_i + (k\lambda_{\min})$ and let the execution enter into a cyclic phase. For all stored iteration states from $\zeta_0$ to $\zeta_{k-1}$, we estimate the throughput based on that state and since we know they are all lower bounds on throughput, we may keep the largest one as the throughput estimate.

*Proposition 3:* The schedule found with the above approximation, leads to a throughput which is no smaller than the approximated throughput.

*Proof:* Assume that the highest approximated throughput $Thr = \frac{1}{\lambda_{min}}$ is obtained from the part of an execution $\sigma_k$ that repeats the part between the $i$th iteration and the $(i+k)$th iteration, resulting in the execution $\sigma = \sigma_i(\sigma_k)^\omega$. Then we can construct an execution $\sigma'$ on the full state space by firing actors in the same order as in $\sigma$. Since $\zeta(\sigma'_{i+nk}) \prec \zeta(\sigma_i(\sigma_k)^n)$, from Proposition 1, we know that for this constructed execution $Thr(\sigma') = \lim_{n\to\infty} \frac{i+n\cdot k}{\|\zeta(\sigma'_{i+nk})\|} \geq \lim_{n\to\infty} \frac{i+n\cdot k}{\|\zeta(\sigma_i(\sigma_k)^n)\|}$ $= \lim_{n\to\infty} \frac{i+n\cdot k}{\|\zeta(\sigma_i)\|+n\cdot k\cdot\lambda_{min}} = \frac{1}{\lambda_{min}} = Thr(\sigma)$. ■

It shows the approximation is conservative.

## VI. Experiments

We implemented our algorithm in the SDF$^3$ toolset [16] and tested it on two sets of RASDFGs to evaluate the iteration-based approach on a Linux 64bit system with an Intel 2.8G

TABLE I
ITERATION BASED APPROACH VS. TIME BASED APPROACH (GRID SEARCH + BOTTLENECK ANALYSIS)

| | Bipartite | | Modem | | Sample Rate | | MP3 | | Satellite | | H263Decoder | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *No. of Shared buffers* | **1** | **2** | **1** | **3** | **1** | **3** | **1** | **3** | **1** | **3** | **1** | **2** |
| *No. of Pareto Points (time based)* | 7 | 8 | 4 | 1 | 6 | 2 | 2 | 4 | 1 | 1 | 2 | 3 |
| *No. of Conf (time based)* | 10 | 69 | 30 | 34 | 15 | 250 | 9 | 104 | 11 | 27 | 65 | 43 |
| *Exec Time time (s)* | 7.8 | 1.7 | 29.7 | 6.7 | 15.3 | 47.9 | 9.3 | 57.2 | 11.4 | 27.4 | 60.9 | 30.9 |
| *No. of Pareto Points (iter based)* | 7 | 8 | 4 | 1 | 3 | 2 | 4 | 2 | 2 | 3 | 1 | 1 |
| *No. of Conf (iter based)* | 10 | 91 | 11 | 34 | 15 | 250 | 9 | 106 | 9 | 9 | 3 | 4 |
| *Exec Time Iter (s)* | 6.7 | 2.2 | 7.2 | 1.7 | 8.9 | 37 | 9.5 | 11.3 | 21.4 | 64.2 | 2.7 | 0.7 |
| $I_\epsilon$*(time,iter)/*$I_\epsilon$*(iter,time)* | 1/1 | 1/1 | 1/1.0625 | 1/1 | 1.024/1.035 | 1/1 | 1.156/1.09 | 1.156/1 | 1.019/1 | 1.019/1 | 1/1.074 | 1/2 |
| *Exec Time Reduction* | 14% | -30% | 75% | 75% | 60% | 23% | -2% | 80% | -87% | -134% | 95% | 98% |

TABLE III
COMBINED APPROACH VS. TIME-BASED AND ITERATION-BASED APPROACHES

| | Bipartite | | Modem | | Sample Rate | | MP3 | | Satellite | | H263Decoder | | Arch 1 | Arch 2 | Arch 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *No. of Shared buffers* | **1** | **2** | **1** | **3** | **1** | **3** | **1** | **3** | **1** | **3** | **1** | **2** | | | |
| *Trade-offs No. (combined)* | 7 | 8 | 4 | 1 | 5 | 2 | 3 | 2 | 2 | 3 | 2 | 3 | 10 | 14 | 18 |
| *Exec Time (combined) (s)* | 14.5 | 3.9 | 36.9 | 8.4 | 24.2 | 84.9 | 18.8 | 68.5 | 32.8 | 91.6 | 63.6 | 31.6 | 153.28 | 66.4 | 150.2 |
| $I_\epsilon$*(iter,comb)/(comb,iter)* | 1/1 | 1/1 | 1.0625/1 | 1/1 | 1.035/1 | 1.041/1 | 1.09/1 | 1/1 | 1/1 | 1/1 | 1.074/1 | 2/1 | 1.092/1 | 1.811/1 | 1.05/1 |
| $I_\epsilon$*(time,comb)/(comb,time)* | 1/1 | 1/1 | 1/1 | 1/1 | 1.024/1 | 1/1 | 1.156/1 | 1.156/1 | 1.019/1 | 1.019/1 | 1/1 | 1/1 | 3.0/1 | 2.0/1 | 2.137/1 |
| *ADRS(time,comb)/(iter,comb)* | 0/0 | 0/0 | 0/0.156 | 0/0 | 0.0056/0.0186 | 0/0.037 | 0.087/0.031 | 0.078/0 | 0.009 /0 | 0.0125/0 | 0/0.0369 | 0/0.0358 | 0.390/0.016 | 0.265/0.310 | 0.303 /0.010 |

TABLE II
ITERATION-BASED APPROACH VS. TIME-BASED APPROACH: PRINTER ARCHITECTURE EXPLORATION

| | **Arch 1** | **Arch 2** | **Arch 3** |
|---|---|---|---|
| *Trade-offs No. (time based)* | 5 | 9 | 9 |
| *Conf No. (time based)* | 147 | 66 | 137 |
| *Exec Time (time based) (s)* | 144.6 | 65.4 | 134.8 |
| *Trade-offs No. (iter based )* | 8 | 5 | 13 |
| *Conf No. (iter based)* | 60 | 6 | 57 |
| *Exec Time (iter based) (s)* | 9.28 | 1 | 15.4 |
| $I_\epsilon$*(time,iter)/(iter,time)* | 3.0/1.092 | 2.0/1.811 | 2.137/1.05 |
| *Exec Time Reduction* | 94% | 98% | 86% |

$Core^{TM}$ i7 with 8GB memory. The first set of graphs includes realistic media graphs from the literature. The set contains a modem [3], a sample-rate converter [3], an MP3 decoder and an H.263 decoder [15], and a satellite receiver [19]. We also included the often used artificial bipartite graph from [3]. Memory resources and corresponding resource requirements are added to the models. For each of the graphs we consider two variants, one with a single shared memory buffer and one with two or three selected shared buffers. The second set of graphs, including resources and resource requirements, originates from an industrial project cooperation with Océ Technologies (www.oce.com) to explore the design space of printer architectures.

We compare the iteration-based approach with the time-based approach of [21], [22]. A bottleneck analysis technique similar to [22] is applied in the iteration-based approach by augmenting time stamps with their producing actor and detecting data and resource dependencies. In our experiments, we divide the resource space into grids and iterate all grid points within a time budget of 1 second for every grid point. Tables I and II show the results. We compare the numbers of explored configurations and the Pareto points obtained. It is not obvious how to compare the quality of sets of Pareto points according to simple metrics [25], [26], [27]. The quality of the Pareto points obtained from both approaches is compared using the $\epsilon$-Indicator of [26] and using the Average Distance to Reference Set (ADRS) [25] metric. The first one is typically used to compare two different point sets while the second one is often used to evaluate the quality of the method to approximate a known Pareto-optimal front.

Loosely speaking, a set $B$ of Pareto points is better than a set $A$ when $I_\epsilon(A, B)$ is bigger than $I_\epsilon(B, A)$. If $I_\epsilon(A, B)$ is bigger than 1, then $B$ contains new Pareto points compared to $A$. $I_\epsilon(A, B)$ defines by how much the points in $B$ need to be scaled so that they are all dominated by points in $A$. Fig. 8 illustrates the $\epsilon$-Indicator. For example, we have to scale by a factor of 1.09 to make that the scaled time-based results are dominated by the iteration-based results. On the other hand, we have to scale by a factor of 1.156 to get the scaled iteration-based results dominated by the time-based results. We can conclude that none of the results are strictly better than the other (both scaling factors are larger than 1), but if we have to make a choice based on the $\epsilon$-Indicator, we would prefer the iteration-based set for this case.

In ADRS, the average distance of a set of Pareto points to the reference set of Pareto points is measured. In our experiment, the combined results of the two approaches are used as the reference set. For example, the 3 green circles in Fig. 8 are the combined results of the two approaches and are used as points of the reference set. For every point in the reference set (green cycle), we compute the minimal distance to the points in the selected set. The distance between two points is defined as the maximal ratio of value change among all objective dimensions. For example, we can compute the ADRS of the iteration-based approach as follows. For two reference points, which overlap with the points in the iteration-based approach, the minimal distance is 0. For the left upper green circle, the minimal distance of a point to it is 0.0929. So the ADRS is $(0.0929 + 0 + 0)/3 = 0.031$.

Since both approaches do not fully explore the design space of large examples, they may miss optimal points. The iteration-based approach cannot exploit the interleaving of iterations, i.e., earlier iterations must use the resources before later iterations. The time-based approach can in principle exploit the opportunity of interleaving iterations. However, in practice, the size of the state space that can be explored is limited and this opportunity typically provides little advantage in practice. As the size of state space grows rapidly with the length of

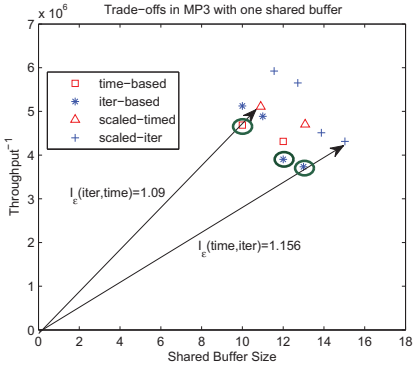



Fig. 8. Trade-offs MP3 with one buffer; circled points are Pareto points for the combined results

an iteration and slows down the exploration, the iteration-based approach typically completes faster than the time-based approach.

We see in our experiments that a considerable exploration time reduction is obtained in 11 out of 15 cases and new Pareto points are found with the iteration-based approach in 8 cases. Also in 8 cases, however, the time-based approach yields Pareto points not found by the iteration-based approach. There does not seem to be a systematic way to predict which of the approaches performs better in specific cases. The two approaches do strengthen each other. Running both analyses yields the best results, as illustrated for example in Fig. 8. Table III compares the combined approach with the two other approaches. The combined approach obviously dominates the two individual approaches quality wise, with acceptable execution times ranging from a few seconds to just over 2.5 minutes.

## VII. Conclusion

In this paper, we investigate a novel technique to explore the resource usage vs. performance trade-offs in dataflow graphs with shared resources using an iteration-based state-space exploration technique. We exploit the properties of the max-plus based model of execution to explore the state space on an iteration-by-iteration basis. We present exact and heuristic pruning techniques to reduce the size of the state space. The experimental results on a set of realistic benchmark models show that the new iteration-based approach and the traditional time-based analysis approach complement each other. The new approach finds new trade-off configurations not found by the traditional technique in 8 out of 15 cases and it is often faster. Combining the two approaches is feasible and yields the highest quality results. The iteration-based approach allows for easy modeling of dynamic execution time changes between iterations, as in scenario-aware dataflow graphs [9], [20]. We plan to investigate the possibilities to add generic resource awareness as developed in this paper to scenario-aware dataflow graphs and adapt our analysis to such a combined model.

## References


[1] F. Baccelli, *et al.*, "Synchronization and Linearity: An Algebra for Discrete Event Systems," *Wiley*, 1992.

[2] F. Balarin, *et al.*, *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer, 1997.

[3] S. S. Bhattacharyya, *et al.*, "Synthesis of embedded software from synchronous dataflow specifications," *J VLSI Signal Process. Syst.*, vol. 21, no. 2, pp. 151–166, 1999.

[4] G. Cohen, *et al.*, "Max-plus algebra and system theory: Where we are and where to go now," *Annual Reviews in Control*, pp. 207–219, 1999.

[5] S. Gaubert, *et al.*, "Modeling and analysis of timed petri nets using heaps of pieces," *IEEE Trans. Aut. Cont*, pp. 40:683–697, 1997.

[6] S. Gaubert, *et al.*, "Performance evaluation of (max,+) automata" *IEEE Trans. Aut. Cont.*, pp. 40:2014–2025, 1995.

[7] M. C. W. Geilen, "Synchronous Data Flow Scenarios" in *ACM Trans. Embedded Computing Systems*, 2010.

[8] M. C. W. Geilen, *et al.*, "Minimizing buffer requirements of synchronous dataflow graphs with model-checking," in *DAC'05 Proc, ACM*, 2005, pp. 819–824.

[9] M. C. W. Geilen, *et al.*, "Worst-case Performance Analysis of Synchronous Dataflow Scenarios" in *CODE+ISSS'10 Proc*, 2010, pp. 125-134.

[10] A. Ghamarian, *et al.*, "Throughput analysis of synchronous data flow graphs," in *ACSD'06 Proc, IEEE*, 2006, pp. 25–34.

[11] B. Heidergott, *et al.*, "Max Plus at Work: Modeling and Analysis of Synchronized Systems" *Princeton Univ. Press*, 2005.

[12] B. Kienhuis, *et al.*, "An approach for quantitative analysis of application-specific dataflow architectures," in *ASAP'97 Proc, IEEE*, 1997, pp. 338–349.

[13] E. A. Lee , *et al.*, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comp.*, vol. 36, no. 1, pp. 24–35, 1987.

[14] P. K. Murthy, *et al.*, "Memory management for synthesis of DSP Software," *CRC Press*, 2006.

[15] S. Stuijk, *et al.*, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC'06 Proc, ACM*, 2006, pp. 899–904.

[16] S. Stuijk, *et al.*, "SDF$^3$: SDF For Free," in *ACSD'06 Proc, IEEE*, 2006, pp. 276–278.

[17] S. Stuijk, *et al.*, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. Comp.*, vol. 57, no. 10, pp. 1331–1345, 2008.

[18] S. Sriram, *et al.*, "Embedded Multiprocessors: Scheduling and Synchronization," *Marcel Dekker, Inc*, 2000.

[19] S. Ritz, *et al.*, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *ICASSP'95 Proc, IEEE*, 1995, pp. 2651–2654.

[20] B. Theelen, *et al.*, "A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis" in *MEMCODE 06*, pp. 185-194.

[21] Y. Yang, *et al.*, "Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs" in *Estimedia'09 Proc*, 2009, pp. 96–105.

[22] Y. Yang, *et al.*, "Automated Bottleneck-Driven Design-Space Exploration of Media Processing Systems" in *DATE'10 Proc*, 2010, pp. 1041–1046.

[23] J. Zhu, *et al.*, "Constrained global scheduling of streaming applications on MPSoCs" in *ASP-DAC'09 Proc*, 2009, pp. 223–228.

[24] J. Zhu, *et al.*, "Pareto efficient design for reconfigurable streaming applications on CPU/FPGA" in *DATE'10 Proc*, 2010, pp. 1035–1040.

[25] J.A. Czyak, *et al.*, "Pareto simulated annealing: a metaheuristic technique for multiple-objective combinatorial optimisation" in *Journal of Multi-Criteria Decision Analysis*, 1998, pp. 7:34–47.

[26] E. Zitzler, *et al.*, "Performance Assessment of Multiobjective Optimizers: An Analysis and Review" in *IEEE Trans, Evol, Comp.*, 2003, pp. 7:117–131.

[27] Gianluca Palermo, *et al.*, "Pareto simulated annealing: a metaheuristic technique for multiple-objective combinatorial optimisation" in *Journal of Embedded Computing*, 2005, pp. 1:305–316.