Mapping of Applications to MPSoCs

Peter Marwedel
TU Dortmund, Informatik 12
Dortmund, Germany

Jürgen Teich
University of
Erlangen-Nuremberg
Erlangen, Germany

Georgia Kouveli
University of
Erlangen-Nuremberg
Erlangen, Germany

Iuliana Bacivarov
ETH Zurich
Zurich, Switzerland

Lothar Thiele
ETH Zurich
Zurich, Switzerland

Soonhoi Ha
Seoul National University
Seoul, Korea

Chanhee Lee
Seoul National University
Seoul, Korea

Qiang Xu
The Chinese University of
Hong Kong, China

Lin Huang
The Chinese University of
Hong Kong, China

ABSTRACT

The advent of embedded many-core architectures results in the need to come up with techniques for mapping embedded applications onto such architectures. This paper presents a representative set of such techniques. The techniques focus on optimizing performance, temperature distribution, reliability and fault tolerance for various models.

Categories and Subject Descriptors

C.4 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Performance, Reliability

Keywords

Embedded Systems, Multi-processor systems on a chip (MPSoCs), application mapping

1. INTRODUCTION

Miniaturization of electronic components has led to the introduction of complex electronic systems which are integrated onto a single chip, so-called systems-on-a-chip (SoCs). At the same time, performance requirements have been increasing. The resulting performance requirements can no longer be met by single processor systems. If achievable at all, this performance would come with an enormous power consumption and serious cooling problems, due to the necessary very high clock speed. Currently, the only option for achieving the necessary performance is to use several processors with moderate clock speeds. Therefore, many of the currently available SoCs contain several processors, making them multiprocessor systems on a chip (MPSoCs). As a result, implementing applications on such a platform includes the mapping of applications onto MPSoCs.

In the general case, this mapping problem is difficult: several use cases have to be considered simultaneously. For each case, a combination of applications must be taken into account to ensure that deadlines or performance guarantees are met. These applications may be using different models of computation [23].

Generated mappings have to include solutions to the scheduling, resource allocation and resource assignment problems. The type and number of resources (processors, buses, etc.) may be either be already specified in the initial input by the designer or they may be generated during the design steps. If this information is not provided by the designer, it has to be generated by solving the **resource allocation** problem. **Resource assignment** is supposed to map computations and communication to hardware resources. Solutions to the **scheduling** problem provide a mapping from operations to the times during which these operations are performed. Multiprocessor scheduling will sometimes already indicate the hardware resources to be used, i.e. the resource assignment problem is included in the scheduling problem. Resource assignment algorithms can be of various types. For example, computations may be permanently mapped to hardware resources.

The ArtistDesign network of excellence (see [2]) identified the application mapping problem as one of the most urgent problems to be solved for implementing embedded systems. A series of workshops on the mapping of applications to MPSoCs was started in order to support moving beyond the state of the art in this area ("Map2MPSoCs", see e.g. [24]). This paper provides a brief overview over contributions of four research groups who presented at the series. These groups were members or affiliates of the ArtistDesign network or gave invited talks.

This paper is structured as follows: section 2 contains a brief summary of related work. In sections 3, 4, 5, and 6,

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

a brief overview of the work of the groups working at the Universities of Erlangen-Nürnberg (Germany), ETH Zürich (Switzerland), the Chinese University of Hong-Kong (China) and Seoul National University (South Korea) is provided. Section 7 will be a brief summary.

2. RELATED WORK

Even though the mapping of applications to MPSoCs is a relatively new area, contributions from other areas can be used as building blocks. Classical scheduling theory for multi-processors provides a mapping from threads to processors and, therefore, results from this area can help to solve the mapping problem. However, scheduling theory typically considers just homogeneous processors whereas results for heterogeneous processors and a corresponding resource allocation are lacking. Classical hardware/software partitioning maps computations to either hardware or software. By mapping to either hardware or software, such partitioning does obviously support the mapping to heterogeneous system components. However, global knowledge about periods, schedulability, etc. is typically not supported. Approaches explicitly designed for mapping to Map2MPSoCs include work presented at the Map2MPSoCs workshop series, such as MAPS [3], DAEDALUS [26] and an approach followed by T. Simunic [28]. Additional related work will be referred to below.

3. RESOURCE-AWARE PROGRAMMING ON TILED ARCHITECTURES

The first approach to application mapping tries to provide programmers with tools for expressing resource requirements of applications. This approach is motivated as follows: In the many-core era, performance will not be the only challenge software developers have to face. Software will also have to guarantee high resource utilization and fault-tolerance and provide power and thermal management. We believe that to facilitate the mapping of applications to the increasing number of available resources, applications will need to be adaptive to a variable resource availability and to their own resource demands. To make that possible, the programmer needs to be provided with the necessary abstractions to express the resource needs of an application, without knowing beforehand the exact details of the underlying architecture. The low-level management of resources should remain the responsibility of run-time system support. Our approach consists of providing the programmer with such a higher level interface to promote the development of adaptive, resource-aware applications. Resource-aware programming is one of the key ideas of *invasive computing* [31, 32], a new paradigm for resource-aware computing that integrates research on MPSoC architectures, compilers, simulation, applications, and run-time support.

3.1 Basics of resource-aware programming

The main idea of resource-aware programming, as the name suggests, is making the application aware of the status and availability of the underlying resources and giving it control over resource allocation and de-allocation. In this way, the application can adjust to changes in degree of parallelism or in the status and availability of resources.

The lifetime of a resource-aware program can be separated in three types of phases: (1) resource allocation, (2) execution, and (3) resource de-allocation. During resource allocation the application requests to allocate resources based on constraints such as number of requested cores, maximum workload or temperature of cores to be allocated, minimum computational capabilities, communication bandwidth or memory size, etc. During the execution phase, the application is allowed to offload computations to the already allocated resources. Last, de-allocation of resources, when the degree of parallelism of the application is lowered, enables better overall resource utilization. These phases can be repeated many times during the lifetime of a resource-aware application, providing the necessary adaptivity. Listing 1 shows what a simple resource-aware application might look like. The application tries to allocate a set of resources, and in case of failure falls back to requesting a different set of resources. Execution is performed on the successfully allocated resources, or sequentially in case both requests failed.

We expect various benefits from resource-aware programming: In addition to improved resource utilization, the performance of individual applications can be improved, as resources that were not available during the application start-up can be allocated later, when they become available. Power consumption can also be improved by dynamic adaptation of voltage/frequency of individual processing elements to match application needs. Monitoring temperature statistics can reduce overheating.

Listing 1: Example pseudocode for a simple resource-aware application

```
// Request resources
claim = request_resources(type, quantity,
                          constraints);
if (successful(claim)):
    execute(claim, code, data);
    release(claim);
else:
    // Try again with different constraints
    claim = request_resources(other_type,
                              other_quantity,
                              other_constraints);
    if (successful(claim)):
        execute(claim, code, data);
        release(claim);
    else:
        // Resource allocation failed again
        execute_sequentially();
```

3.2 Application on Tiled Architectures

Motivation: Resource-aware programming is of interest when it comes to future many-core architectures, but we can also evaluate potential benefits on existing multi-core architectures. For this purpose, we focus in this paper on Tilera's 64-core TILEPro64 processor [36] and Intel's 48-core Single-Chip Cloud Computer (SCC) [25].

These two architectures are both based on processor tiles and use a large number of processing elements. They consist of a number of tiles replicated on a single chip, interconnected with a network-on-chip (NoC).

Tilera's TILEPro64 offers coherent shared memory, thus allowing the use of wide-spread shared memory programming approaches. Two of the on-chip networks, one static and one dynamic, are also accessible at user-level for sending short messages [36]. Intel's SCC, however, lacks hardware support for memory coherence. The main programming models used with this processor are based on message

passing, which is implemented over the on-chip non-coherent shared memory, named message passing buffer (MPB) [25].

Current operating systems do scale well and exhibit problems for these architectures. On TILEPro64, SMP Linux is used, which cannot provide the level of scheduling and mapping support that competing applications need on such a number of cores.

Implementation: To make an early evaluation of resource-aware programming for these architectures, we built a library to support its basic concepts at user-level. For our initial implementation, the resources we consider are the cores of each processor, of which applications request exclusive use. Due to the differences of system software support on the two processors, our implementations of these libraries will also vary. Our goal, however, is to provide a common interface for resource-aware programming, to be used with both architectures. Listing 2 shows what such a common API could look like, and is actually the API that we have currently implemented for TILEPro64 and partly implemented for SCC.

Listing 2: API for resource-aware applications on tiled architectures

```
// Initialize - finalize resource-aware application
void resource_aware_app_init();
void resource_aware_app_finalize();

// Allocate cores
// num:    number of requested cores
// claim:  set of CPUs successfully allocated
int request_cpus(int num, cpu_set_t *claim);

// Allocate grid of cores
// h, w:   height and width of requested core grid
// claim:  set of CPUs successfully allocated
int request_grid(int h, int w, cpu_set_t *claim);

// Release cores
// claim: set of cores to release
int release_cpus(cpu_set_t claim);

// Execute function on allocated cores:
// claim: set of allocated cores
// start_routine: array of function pointers
// arg: arguments, ret: return values
int execute(cpu_set_t claim, int ncpus,
            void*(*start_routine[])(void *),
            void *arg[], void **ret[]);

// Execute program on allocated cores:
// claim: set of allocated cores
// path: array of program names
// argv: array of arguments
int execute_cpus(cpu_set_t claim, const char *path,
                 char *const argv[]);
```

For the implementation of such an API, we have to consider the partitioning of resource information among the cores, as well as the distribution of responsibility of servicing resource allocation requests.

Since Tilera TILEPro64 offers support of coherent shared memory, the simpler choice is to keep resource status information in shared memory, and to provide resource-aware applications access to that shared information through library calls. This centralized approach leads to improved resource utilization, since all information on resource status is available when making resource allocation decisions. On the other hand, the shared information is a bottleneck and performance gets worse when increasing the number of processes competing for resources. For our initial implementation we chose to use this approach, but we plan to look into distributed or hierarchical approaches to resource management as well [19].

For enabling resource-aware programming on Intel's SCC processor, we follow a more distributed approach, having a process to service allocation requests executed on each core. The resource availability information is stored in the non-coherent shared on-chip memory. This more distributed approach has the benefit that there exists no central bottleneck, but applications lack a general view of status and availability of resources. Thus, resource allocation decisions will tend to be suboptimal, and requests such as allocating a grid of cores will be more time-consuming to service.

There are some basic challenges yet to be faced for SCC. As a different operating system instance is operating on each core, the execution of arbitrary pieces of code on another core is complicated. Currently, we only support execution of full programs on remote allocated cores, but we are working on also executing code of the granularity of functions implementing a given interface instead of full programs. For this reason, we currently have two separate versions of `execute()` in our API (see Listing 2).

3.3 Experimental Results

Table 1 shows the overheads of the three basic constructs of resource-aware programming, as measured when a single application is running on each processor.

Table 1: Overheads of a single resource-aware application (in cycles).

cores	allocate		execute (in 10^3)		release	
	Tilera	SCC	Tilera	SCC	Tilera	SCC
1	1147	1119	2708	8848	11381	1393
2	6269	1694	3797	6154	10232	1408
4	6402	2789	6332	6358	11084	2114
8	6904	5119	10165	40215	12439	3449
16	7480	10090	24411	41531	13298	6145
32	9267	19656	62404	6965	15758	12673

To compensate for the measurements' variability due to the interference of the operating system, we calculate the average of 100 executions. In order to isolate the overhead of the execution phase (starting a new thread/process, joining etc.) from the execution time of the particular application, we use a function which only returns zero on TILEPro64, and a program which only returns zero on SCC. This overhead is to be compared with the execution time (per thread) of the application, to determine the granularity at which it makes sense to parallelize and still expect gains in real-world applications.

For TILEPro64, offloading execution to an allocated core incurs an overhead in the order of millions of cycles, which rises almost linearly as the number of cores increases. Therefore, our current approach would not be appropriate for fine-grained parallelization. This high overhead is due to the process/thread creation time, which is a costly operation. To reduce this overhead, alternative implementation approaches need to be considered.

The high execution overhead can also be noticed on SCC, although in that case it is more variable and does not have a clear correlation to the number of cores. This results from the distributed design of our library. Each new process is started by the "server" process running on each core, thus distributing the overhead. The results presented here are preliminary, as this is is ongoing work.

4. REAL-TIME AND TEMPERATURE AWARE MAPPING OPTIMIZATION

The increase in performance of modern real-time MPSoC comes with an increase in power density and high on-chip temperatures, which in turn may decrease the reliability and performance of the system. Embedded systems like cell phones or more sophisticated 3D architectures cannot afford sufficient cooling or cheap cooling solutions. One way to address these challenges is to optimize the system design for both worst case performance and worst case temperature. This requires ruling out mapping alternatives that do not conform to real-time and peak temperature requirements of the system already in early design stages. In this section, we present the distributed operation layer (DOL) [33] as a mapping optimization framework that considers both performance and thermal characteristics of MPSoCs.

4.1 Mapping Optimization Loop

Figure 1 illustrates the task of mapping optimization as it is considered in DOL [33]. Implementing the Y-chart design paradigm [17], DOL considers parallel streaming applications represented as process networks and specified independently from the distributed memory architecture. An optimized mapping is found after the system-level exploration of different design alternatives. Mapping refers to both binding application elements to computation and communication resources, and scheduling on shared computation or communication resources. Ultimately, the same system-level specification of the application and architecture together with the optimal mapping specification form the synthesizable system specification, which will be implemented on the final system or can be simulated on the virtual platform. Typically, we use low level simulation in a feedback loop for automatically calibrating our time and thermal analysis models used for comparison of different mapping alternatives [9] [10] [34].

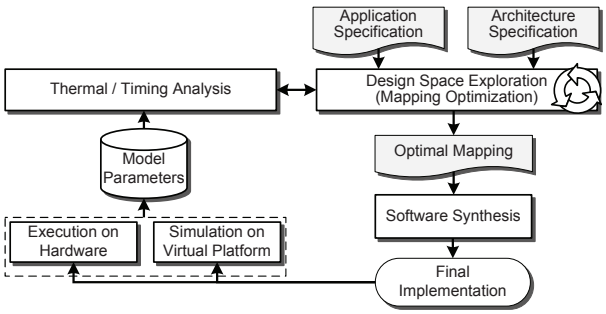


Figure 1: Distributed operation layer (DOL) real-time and thermal-aware mapping optimization.

4.2 Real-Time Analysis

To analyze timing properties and provide real-time guarantees on investigated systems, DOL integrates the modular performance analysis (MPA) [35] [4].

In DOL, the MPA model is generated and calibrated based on the same system specification as used for software synthesis. The model preserves an accurate representation of the system, but it requires properly estimated parameters. The automated MPA model generation and calibration method is described in [9] and [10].

Figure 2 shows an MPA representation of a basic system with three processes mapped on two processors that communicate via a shared bus. Concretely, an MPA model is composed of abstract elements representing (a) the computation and communication of an application, (b) resources such as processors and interconnects, (c) resource sharing methods, and (d) event streams that carry data and trigger actors. The approach is based on real-time calculus (RTC) [5], an extension of network calculus [20], that uses arrival curves $\alpha(\Delta)$ for event streams, service curves $\beta(\Delta)$ for resource availability, and derived workload curves $\gamma(\Delta)$.

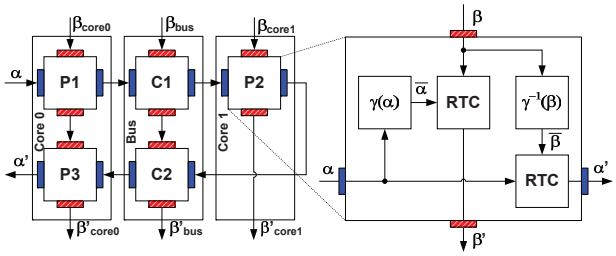


Figure 2: MPA model of a system mapping three communicating processes on two cores connected via a bus; preemptive fixed priority scheduling is used on all resources.

In the MPA abstraction, a component receives in the time interval $[s,t)$ a cumulative workload (trace) of $R(s,t)$ time units, upper-bounded by an upper arrival curve α, where

$$R(s,t) \leq \alpha(t-s) \qquad \forall s < t \tag{1}$$

with $\alpha(0) = 0$. For work-conserving scheduling, the accumulated computing time $Q(s,t)$ of the component in time interval $[s,t)$ is $Q(s,t) = \inf_{s \leq u \leq t}\{(t-u) + R(s,u)\}$. $Q(s,t)$ is upper bounded by the workload curve $\gamma(t-s)$, written as

$$Q(t-\Delta, t) \leq \gamma(\Delta) = \inf_{0 \leq \lambda \leq \Delta}\{(\Delta - \lambda) + \alpha(\lambda)\}. \tag{2}$$

4.3 Worst-Case Thermal Analysis

Similar to worst-case timing analysis in MPA, we aim to bound the worst-case peak temperature, that is, the maximum temperature that can occur under all possible scenarios of task executions. In real-time analysis, for work-conserving real-time scheduling algorithms, the time critical instant is to release all tasks/events as early as possible without violating arrival curve constraints [5].

For temperature analysis, the temperature critical instant for a single processor is identified among infinitely many traces that comply with the event stream specification in MPA, in [27]. We demonstrate that the workload trace that leads to the worst-case peak temperature at time instant τ is to release events in time interval $[\tau - \Delta, \tau)$ such that the processed computation in time interval $[\tau - \Delta, \tau)$ is maximized under arrival-curve constraints. For example, for periodic tasks with jitter, the worst-case execution time occurs if burst arrivals and jitters happen at system initialization, while the worst-case peak temperature occurs if the system is first warmed up with periodic arrivals and then suddenly heat up with burst arrivals and jitters. Therefore, $R^*(0,\Delta) = Q^*(0,\Delta) = \gamma(\tau) - \gamma(\tau - \Delta)$, with $\gamma(\Delta) = \inf_{0 \leq \lambda \leq \Delta}\{(\Delta - \lambda) + \alpha(\lambda)\}$ leads to worst-case temperature,

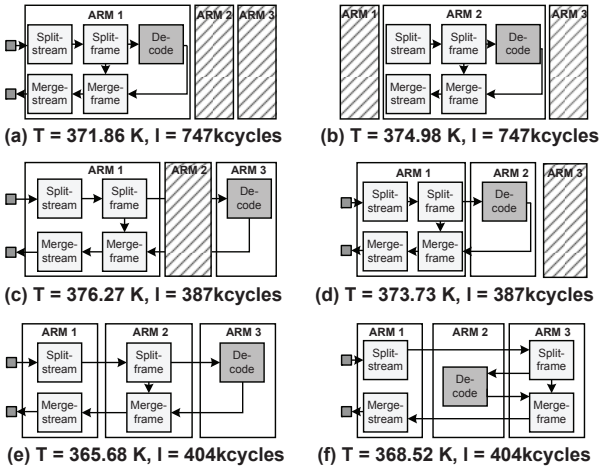


Figure 3: Trade-off between worst case latency and peak temperatures, depending on the mapping/-placement of MJPEG process network on 3 cores.

among all possible traces. The upper bound on component temperature $T^*(\tau)$ is guaranteed at time τ, with a precision determined by steady-state temperature and thermal parameters of the underlying hardware, see [27].

However, for multi processor systems, besides the worst-case temperature due to self-heating described above, heat transfer between components needs to be equally considered. In addition to the upper bound on the temperature of each processing component k, $T_k^*(\tau)$, we have to construct the worst-case temperature sequence ${Q_i}^*(0, \Delta)$ for all neighbor components $i \in \{1, n\}$ with respect to the chosen component k. The worst-case neighboring sequences will differ for all analyzed components due to geometrical differences and imply different transfer delays between cores. Finally, the worst-case peak temperature of the system T_S^* is computed as the maximum of all component temperatures T_k^*, that is, $T_S^* = \max(T_1^*, \ldots, T_n^*)$.

Often, finding the exact solution on worst-case heat transfer is only possible via exhaustive search due to the huge number of possibilities in event patterns and undetermined locations of heat transfer functions maxima. To speed up the calculations, we practically over-approximate neighboring temperatures by simply maximizing the convolution between mirrored burst activities and accumulated maximum period of activity in the vicinity of the approximated maximum heat transfer function. In our experiments, the over-approximation comes with a very small error of below 1% in temperature variation.

4.4 Illustrative Results

In this section, DOL is used to optimize the mapping of a motion JPEG (MJPEG) decoder on three ARM cores communicating via a shared bus. Both worst-case latency and worst-case peak temperature are considered during optimization. The MPARM virtual platform [1] is used to determine the computational demand of different processes and the power dissipation of ARM cores. Fixed priority preemptive scheduling is used on all processors, while a TDMA policy is employed on the bus. The application is driven by a periodic input stream, with a period of 110 kcycles and a jitter of 220 kcycles. Thermal parameters are obtained from HotSpot [29].

Trade-off solutions on one, two, and three cores are illustrated in Figure 3. The processors are identical, but they have different thermodynamical properties, due to their placement.

Solution pairs where only the placement the homogeneous processors has been changed (e.g., mappings a and b), indicate that physical placement cannot be ignored in temperature analysis. In our simple examples, we observe the same latency in all pairs because of the homogeneous processor structure and the symmetric shared bus, but more than 3K maximum temperature variation.

No solution is optimal with respect to both latency and temperature. The load balanced mapping (mappings e and f) is a "cool" solution, due to small neighboring heat transfer and shorter accumulated bursts. The two-processors solution (mappings c and d) has lower latency since it eliminates some of the inter-processor communication overhead, but it is characterized by the highest peak temperature due to neighboring effect and long accumulated bursts. Note, that mappings on fewer cores could present lower temperatures if unused processors would be turned off. In our experiments, unused processors still consume (non-negligible) idle power.

In conclusion, DOL guides the MPSoC mapping considering the non-trivial dependency between performance and maximum temperature (and consequently reliability). Our analytic models can guarantee the final performance and correct function of the system, considering both functional and non-functional properties.

5. AGING-AWARE MAPPING OF APPLICATIONS TO MPSOCS

The third approach to application mapping focuses directly at lifetime reliability of MPSoCs, including potential wearout failures. If the wearout is not taken into consideration during the task allocation and scheduling (TAS) process, some processors might age much faster than the others and become the reliability bottleneck for the embedded system. Most prior work in reliability-driven TAS solutions assumes processors' failure rates to be constant values (i.e., independent of their usage times), and thus is unable to capture the system's accumulated aging effects.

To tackle the above problem, we proposed the first work that explicitly takes lifetime reliability into consideration during the TAS process [14]. We propose an analytical model to estimate the lifetime reliability of multiprocessor platforms when executing periodical tasks, and we present a novel lifetime reliability-aware task allocation and scheduling algorithm based on simulated annealing technique. In addition, to speed up the annealing process, several speedup techniques are proposed to obtain an approximated mean time to failure (MTTF).

In the above work, for the sake of simplicity, only a single execution mode is considered. However, today's complex embedded systems usually work across a set of different interacting applications and operational modes. Motivated by the above, in [12], we showed how to conduct energy-efficient task allocation and scheduling on MPSoC platforms for such multi-mode embedded systems, taking the lifetime reliability as a constraint. Since the lifetime reliability constraint is a system-wide constraint, it is not necessary to ap-

ply the same reliability constraint to every execution mode for multi-mode MPSoC embedded systems. We first generate "good" solutions in terms of reliability and/or energy for each execution mode and then search for an optimal combination of these to obtain minimized energy while satisfying the system-wide lifetime reliability constraint.

Customers of electronic products may have different usage strategies for the same system. This is referred as the *usage strategy deviation* of the product. Because of this, a unified task schedule for every execution mode generated at design-stage may not be reliable or energy-efficient. In [13], we proposed a novel customer-aware task allocation and scheduling technique to tackle this problem, wherein initial schedules are generated at design stage and each product is optimized separately with online adjustment at regular intervals.

5.1 Problem Formulation

Based on the above, the problem of aging-aware mapping of applications to MPSoCs can be formulated as follows:

Problem: Given

- q execution modes. A directed acyclic task graph $\mathcal{G}^k = (\mathcal{T}^k, \mathcal{E}^k)$ for mode k, wherein each node in $\mathcal{T}^k = \{\tau_i^k : 1 \leq i \leq n^k\}$ represents a task in $\mathcal{G}^k$, and $\mathcal{E}^k$ is the set of directed arcs which represent precedence constraints. Each task graph $\mathcal{G}^k$ has a deadline d^k;
- The joint probability density function that the system is in various modes $f_{\mathbf{Y}}(\mathbf{y})$, where y_k represents the probability that the system is in execution mode k;
- A platform-based MPSoC embedded system that consists of a set of processors $\mathcal{P} = \{P_j : 1 \leq j \leq m\}$, belonging to $\mathcal{V}$ categories;
- Execution time table $\{c_{k,i,j} : 1 \leq k \leq q, 1 \leq i \leq n^k, 1 \leq j \leq m\}$, where $c_{k,i,j}$ is the execution time of task τ_i^k on processor P_j. If a task cannot be executed by a certain processor, the corresponding $c_{k,i,j}$ is set to infinity;
- Power consumption table $\{p_{k,i,j} : 1 \leq k \leq q, 1 \leq i \leq n^k, 1 \leq j \leq m\}$, where $p_{k,i,j}$ is the power consumption of τ_i^k on processor P_j;
- The target service life t_L and the corresponding reliability requirement $\eta\%$;
- Failure mechanism parameters (e.g., activation energy E_a of electromigration) and the corresponding failure distributions;

To determine a periodical task schedule for each execution mode such that the objective (e.g., energy consumption) is optimized under specified system requirements, such as the performance constraint that all tasks are finished before deadlines and the reliability constraint that the expected reliability at the target service life is no less than $\eta\%$.

5.2 Lifetime Reliability Estimation

As suggested in JEP85 [16], we use a Weibull distribution to describe the wearout effects for MPSoCs. Since the slope parameter is shown to be nearly independent of temperature, the reliability of a single processor at time t can be expressed as

$$R(t,T) = e^{-(\frac{t}{\alpha(T)})^{\beta}} \quad , \tag{3}$$

where T, $\alpha(T)$, β represent temperature, the scale parameter, and the slope parameter in the Weibull distribution, respectively. Processors' operational temperatures vary significantly with different applications. Typically, when a processor is under usage or its "neighbors" on the floorplan are being used, its temperature is higher than otherwise. Therefore, instead of assuming T as a fixed value, we consider the temperature variations in our analytical model for more accuracy. Other factors that affect a processor's lifetime reliability are also considered in the model. Architecture properties of processor cores are reflected on the slope parameter β, while the cores' various operational voltages and frequencies manifest themselves on $\alpha(T)$.

Our analytical framework takes the hard error models as inputs, and hence it is applicable to analyze any kind of failure mechanism, including the combined failure effects shown in [30]. For the sake of simplicity, we take electromigration failure mechanism as an example in our works. The scale parameter is

$$\alpha(T) = \frac{A_0 (J - J_{crit})^{-n} e^{\frac{E_a}{kT}}}{\Gamma(1 + \frac{1}{\beta})} \quad , \tag{4}$$

where A_0 is a material-related constant, $J = V_{dd} \times f \times p_i$ [6], and J_{crit} is the critical current density.

The details of this lifetime reliability model, which captures the accumulated aging effects of IC hard errors, have been illustrated in [14]. We resort to this model to estimate the reliability stress of task schedules. Let $\alpha_k(T)$ be the aging effect of processor k in unit time provided operational temperature T. The aging effect of processor k in a certain task schedule is therefore

$$\begin{aligned} A_{i,k} &= \sum_j \Big[\frac{w_{i,j,k}(\rho) \cdot 1_{\{i,j \to k\}}}{\alpha_k(T_{i,j,k})} - \frac{w_{i,j,k}(\rho) \cdot 1_{\{i,j \to k\}}}{\alpha_k(T_{amb})} \Big] \\ &\quad + \frac{d_i}{\alpha_k(T_{amb})}, \end{aligned} \tag{5}$$

where, T_{amb} is the ambient temperature. Then, suppose the system remains in mode i through its service life, we can express system reliability at the target service life L with the aging effect additivity proved in [14] as

$$R_i = \exp\Big[- \sum_k \big(\frac{A_{i,k}}{\max_j\{d_{i,j}\}} \cdot L \big)^{\beta_k} \Big] \quad . \tag{6}$$

5.3 Proposed TAS Technique

Various optimization techniques have been proposed in the literature to address the TAS problem. In our works, we used a simulated annealing (SA)-based algorithm. An efficient solution representation together with the corresponding moves is very importasnt for any SA-based technique. To the best of our knowledge, we propose the first solution that has a 1-to-1 correspondence between the representation and the task schedule. Please refer to [13] for more details.

We mainly minimize the expected energy consumption under performance and reliability constraints. The cost function used in our SA-based technique consists of three terms, each for one objective or constraint, as below,

$$\begin{aligned} \text{Cost} &= \mathbf{E}_{\mathbf{Y}}[\text{Energy}] + \mu \cdot \mathbf{I}[\exists i, k : e_i^k > d^k] \\ &\quad + \mu \cdot \mathbf{I}[R^{sys}(t_L) < \eta\%], \end{aligned} \tag{7}$$

where $\mathbf{E}_{\mathbf{Y}}$ indicates the expectation over usage strategy distribution $\mathbf{Y}$, μ is a significantly large number, and the indicator function $\mathbf{I}[A]$ equals to 1 if event A is true while 0 otherwise. Thus, the large cost μ is the penalty of violating

constraints. In case that no DVFS technique is enabled, the computation of energy consumption in an execution mode k is a trivial problem. We can simply sum up the energy consumption of all tasks. Given the task schedule, the second term in Eq. (8) can be determined by comparing the finish time e_i^k and deadline d_i^k of tasks and it is independent of usage strategy. As for the third term, with the similar argument described in Section 5.2 and detailed in [12, 11], we can obtain the expected aging rate and approximate the reliability function $R^{sys}(t_L)$.

5.4 Experimental Results

The effectiveness of the proposed technique is demonstrated as follows: First, task graphs are generated by TGFF [7] and scheduled on hypothetic MPSoCs. The power consumption values are randomly generated while the range is set in agreement with the parameters published for IBM's PowerPC 750CL [15]. Although the proposed analytical model is applicable for any failure mechanisms or their combinations, because of the lack of public data on the relative weights of different failure mechanisms, we consider a widely-used electromigration model [8] in our experiments.

To validate the effectiveness of our approximation for MTTF, we take the valid task schedules obtained from our algorithm and compute the approximated $MTTF$. Then, we derive the accurate $MTTF$ values by monitoring the temperature variation using HotSpot for the same schedules and compare them to the approximated values. As shown in Fig. 4 for a homogeneous 2-processor platform, our approximation is able to reflect the quality of different solutions. That is, if a schedule has larger mean time to failure, it tends to have a larger approximated value.

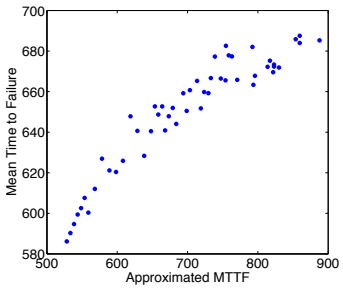


Figure 4: Comparison between Approximated $MTTF$ and Accurate Value.

In [14], we try to maximize the lifetime reliability of MPSoCs under performance constraints. The results obtained by using our algorithm have a much longer lifetime compared to a baseline thermal-aware scheduling algorithm, especially when we can relax the schedule constraints. For instance, when we relax the task deadline by 5%, the lifetime improvement is 12.17% on a homogenous 6-processor platform and 41.93% on a heterogeneous 6-processor platform.

In [12], we conduct TAS for multi-mode MPSoCs to optimize energy consumption under lifetime reliability constraints. For the experimental design, our multi-mode solution is able to achieve low energy consumption meeting the lifetime reliability requirement, while the greedy heuristic approach cannot provide a solution meeting this requirement. In addition, the multi-mode method is able to achieve 14.1% energy savings when compared to the earlier single-mode approach.

In [13], a customer-aware task allocation and scheduling technique is presented to adapt to the specific usage strategy of individual products. We plot the measurements of all sample products in terms of energy consumption and lifetime reliability in Fig. 5. As can be observed from this figure, for those chips that have larger reliability margin, the proposed online adjustment leads to more energy reduction. In contrast, for the other chips we achieve reliability enhancement by sacrificing some energy consumption.

6. FAULT-AWARE TASK SCHEDULING UNDER REAL-TIME CONSTRAINTS

The fourth approach to mapping reflects the fact that processors might fail, despite temperature- and reliability-aware application mapping. A traditional solution to tolerate a processor failure is to use resource redundancy such as physical hardware replication or multiple software versions [18]. However, the adoption of redundancy is often avoided for resource constrained embedded systems. Another approach is to migrate the tasks on the faulty processor to other live processors. Previous work on migration mostly focuses on minimizing the overhead of task migration [22].

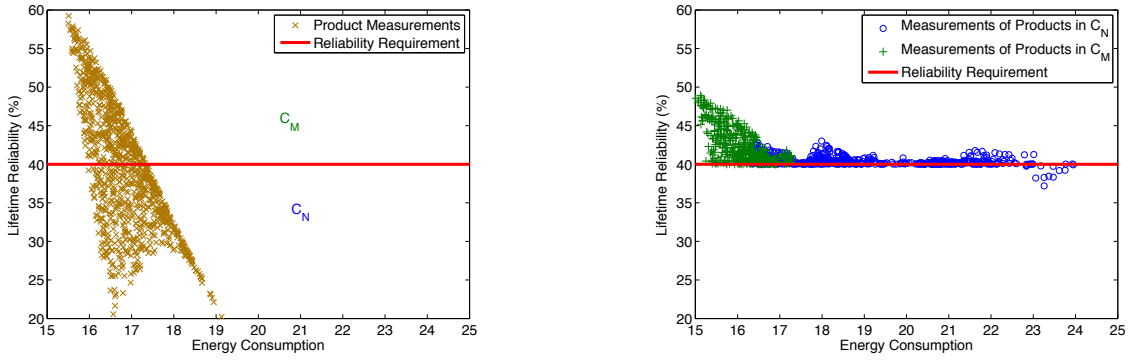
If an application is specified by a decidable dataflow graph, SDF (synchronous dataflow) [21], or CSDF (cyclo-static dataflow) graph, it is possible to construct a static schedule that considers the possibility of processor failures.

However, we are concerned about throughput-oriented applications that may also have a latency constraint. For those applications, we propose a novel technique, called a re-scheduling technique, that re-schedules the task graph to maximize the throughput under a given latency constraint for each fault scenario. If a fault is detected at run-time, the live processors fetch the saved schedule, perform migration, and execute the remaining tasks of the current iteration. We consider the migration overhead when constructing a static schedule for each failure scenario. Since a fault may occur on any processor in any time, we have to consider the worst-case scenario to guarantee satisfaction of the latency constraint. Since the scheduling problem is no easier than an NP-hard problem of simple multiprocessor scheduling, we use a genetic algorithm to obtain a near-optimal re-scheduling result for each fault scenario.

In addition, the basic migration policies, preemptive and non-preemptive, are compared. When a fault is detected, the preemptive policy interrupts the current task and starts the re-scheduling step immediately. In contrast, the non-preemptive policy waits until the current task finishes, and then starts the re-scheduling step. We investigate the effect of the migration policy on the latency of the current iteration, and propose a hybrid policy to obtain better performance.

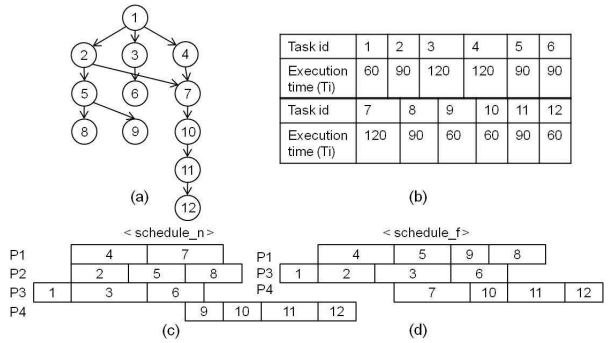
6.1 Problem Definition

A homogeneous SDF graph is shown in figure 6(a) for simple illustration. We assume that the task execution time is given a priori on each processor core, as shown in figure 6(b). We also assume that an initial schedule which maximizes the throughput is given for all available processor cores. Figure 6(c) displays an example schedule on four processors ignoring the communication overhead for simplicity. The target architecture can be any heterogeneous multi-processor system onto which a static schedule can be



(a) Initial Solution (b) Online Adjustment

Figure 5: Comparison of Initial Solution and Online Adjustment in Product Measurements.



Task id	1	2	3	4	5	6
Execution time (Ti)	60	90	120	120	90	90
Task id	7	8	9	10	11	12
Execution time (Ti)	120	90	60	60	90	60

Figure 6: (a) An example task graph, (b) task execution times, (c) schedule before a fault occurs, and (d) schedule after a fault occurs on P2

found. While a fault can occur at any moment on any processor during execution, we assume that the process failure is checked at the task boundary. Then a fault scenario is uniquely defined by a task that encounters a fault during execution. The total number of fault scenarios is the total number of task invocations in an iteration of the input graph. In the proposed technique, we consider all possible fault scenarios. For instance, suppose that a fault occurs on P2 while task 2 is executed. Then we re-schedule the task graph onto the remaining live processors P1,P3,P4 to maximize the throughput, which is displayed in Figure 6(d).

To compute the latency, we have to determine when to perform task migration. Figure 7 illustrates three cases assuming that the processor failure is detected and notified at the completion time of task 2. In case we perform migration immediately after a fault is detected (preemptive policy), we stop task 3 on P3 and task 4 on P1 in the middle of execution. Then each processor fetches the schedule of figure 6(d) and performs task migration accordingly. Afterwards, it executes the tasks that have not been completed in the current iteration, following the fetched schedule after failure.

The second case of figure 7 shows another policy, called non-preemptive policy, where task migration is delayed until the current task is completed. In this example, the non-preemptive policy shows better performance than the preemptive policy in terms of latency. We may use a hybrid policy that applies both policies selectively. In the last case of figure 7, task 3 on P3 is preempted but task 4 on P1 is not. We can find a hybrid policy that gives the best result at each fault scenario.

In summary, we aim to find a static schedule and a migration policy at each fault scenario, that maximize the throughput satisfying a given latency constraint. By varying the latency constraint, we could obtain the pareto-optimal solutions.

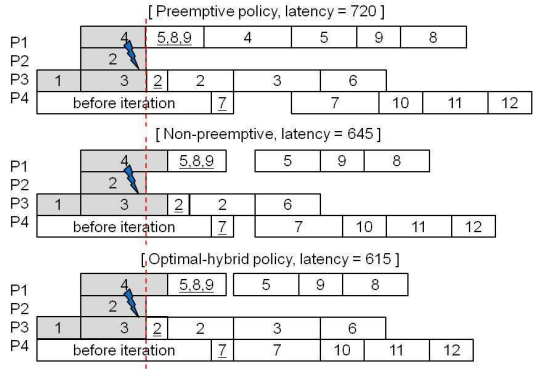


Figure 7: Latency computation for three migration policies

6.2 Proposed Task Re-scheduling Technique

The proposed technique consists of two parts; intensive compile-time analysis to find schedules that maximize the throughput with live processors for all fault scenarios, and run-time management to migrate tasks and resume the execution after fetching the saved schedule. The compile-time analysis flow is outlined in figure 8.

For each fault scenario, we use a GA(genetic algorithm)-based heuristic to find a static schedule that maximizes the throughput under a latency constraint. A candidate solution represents the mapping and scheduling information of all tasks on the live processors. If a hybrid migration policy is chosen, the migration policy of each task is also encoded in the candidate solution. For each candidate solution, we first compute the worst-case latency of the application by simulating the schedule as illustrated in figure 7. We filter out the solutions that fail to meet the latency constraint

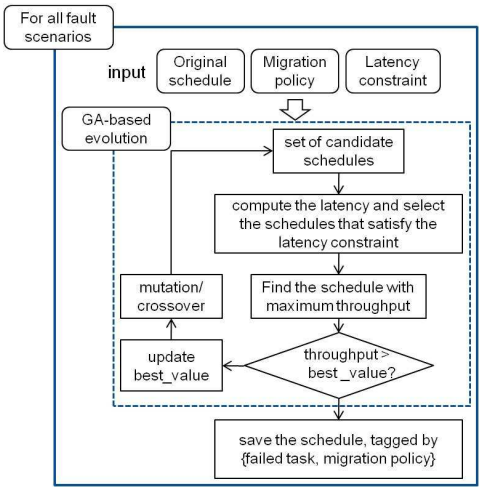


Figure 8: The compile-time analysis flows

Table 2: Task graphs used for experiments

Graph	1	2	3	4	5
Number of procs	3	4	3	8	8
Number of tasks	8	12	24	40	50

and choose the best schedule that maximizes the throughput. We terminate the evolution process in case there is no improvement of the throughput or the user-defined limit on the number of evolution cycles is reached. Note that we can obtain the Pareto-optimal solutions by varying the latency constraints and performing the analysis repetitively.

6.3 Experimental Results

For experiments, task graphs are randomly generated and the execution times of tasks are assigned randomly with 300% variation. The migration cost of each task is set to 50% of its execution time. Table 2 summarizes the number of tasks in the task graph and the number of processors used for each experiment.

In the first set of experiments, we compare the latencies of the same schedule that maximizes the throughput ignoring the latency constraint. It is observed that the hybrid policy reduces the latency by up to 15% compared with the other policies while there is no preference between preemptive and non-preemptive policies. As the second set of experiments, we find a throughput-maximized schedule by varying latency constraints. Figure 9 shows the results for graph 1 experiment, where three migration policies are compared. The x-axis represents the latency constraint and the y-axis represents the resultant throughput normalized to the initial throughput before a fault occurs. Three groups are distinguished depending on which processor fails. The graph shows that the throughput degrades as the latency constraint is tightened. In some cases, only the hybrid policy could find a solution.

7. CONCLUSION

The preceding sections have clearly stated the improvement over the state of the start as it existed a few years ago. Tools like the ones that were presented consider several constraints and objectives. It is now possible to map combinations of applications to MPSoCs, while performing multi-objective optimization. The trend toward addressing more objectives continues. An integration of all types of optimizations into a single tool would, however, make that tool very complex. It would become even more complex if the union of execution platforms and applications models would need to be supposed. Nevertheless, tools like the ones that were presented should soon become mature enough to be used in really large industrial applications.

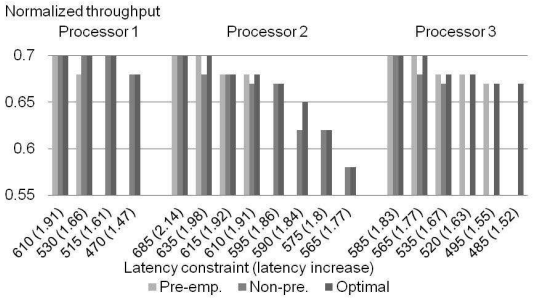


Figure 9: Throughput variation according to the latency variation for graph 1 experiment

The authors would like to thank the large number of institutions supporting this work.

8. REFERENCES

[1] Luca Benini, David Bertozzi, Bogliolo Alessandro, Francesco Menichelli, and Mauro Olivieri. MPARM: exploring the multi-processor SoC design space with SystemC. *The Journal of VLSI Signal Processing*, 41:169–182(14), 2005.

[2] B. Bouyssounouse. Home page of the ArtistDesign network of excellence. *http://www.artist-embedded.org*.

[3] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. MAPS: an integrated framework for MPSoC application parallelization. In *45th annual Design Automation Conference (DAC)*, pages 754–759, 2008.

[4] Samarjit Chakraborty, Wolfgang Haid, Kai Huang, Simon Künzli, Alexander Maxiaguine, Simon Perathoner, Tobias Rein, Nikolay Stoimenov, Lothar Thiele, and Ernesto Wandeler. Modular performance analysis and real-time calculus. http://www.mpa.ethz.ch, 2009.

[5] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analyzing system properties in platform-based embedded system design. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, 2003.

[6] A. Dasgupta and R. Karri. Electromigration reliability enhancement via bus activity distribution. *Proc. DAC*, pages 353–356, 1996.

[7] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task graphs for free. *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, pages 97–101, 1998.

[8] A. K. Goel. High-speed VLSI interconnections. *IEEE Press*, 2007.

[9] Wolfgang Haid, Matthias Keller, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Generation and calibration of compositional performance analysis models for multi-processor systems. In *Proc. Intl Conference on Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 92–99, Samos, Greece, 2009. IEEE.

[10] Kai Huang, Wolfgang Haid, Iuliana Bacivarov, Matthias Keller, and Lothar Thiele. Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. *ACM Transactions in Embedded Computing Systems (TECS)*, 2011.

[11] L. Huang and Q. Xu. Agesim: A simulation framework for evaluating the lifetime reliability of processor-based socs. *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, pages 51–56, 2010.

[12] L. Huang and Q. Xu. Energy-efficient task allocation and scheduling for multi-mode MPSoCs under lifetime reliability constraint. *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, pages 1584–1589, 2010.

[13] L. Huang, R. Ye, and Q. Xu. Customer-aware task allocation and scheduling for multi-mode MPSoCs. *Proc. ACM/IEEE Design Automation Conference (DAC)*, 2011.

[14] L. Huang, F. Yuan, , and Q. Xu. Lifetime reliability-aware task allocation and scheduling for MPSoC platforms. *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, pages 51–56, 2009.

[15] IBM. IBM PowerPC 750CL Microprocessor Revision Level DD2.x. *http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/2F33B5691BBB8769872571D10065F7D5/$file/750cldd2x_ds_v2.4_pub_29May2007.pdf*, 2007.

[16] Joint Electronic Device Engineering Councils (JEDEC). Jesdec85: Methods for calculating failure rates in units of fits. *JEDEC Publication*, 2001.

[17] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proc. of the Int'l Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 338–349, Washington, DC, USA, July 1997.

[18] I. Koren and C. M. Krishna. *Fault-tolerant systems*. Morgan Kaufmann Publisher, 2007.

[19] G. Kouveli, F. Hannig, J.-H. Lupp, and J. Teich. Towards Resource-Aware Programming on Intel's Single-Chip Cloud Computer Processor. In *Proceedings of the 3rd MARC Symposium*, Ettlingen, Germany, July 2011. KIT Scientific Publishing.

[20] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus — a theory of deterministic queuing systems for the internet*, volume 2050 of *LNCS*. Springer Verlag, Berlin, Germany, 2001.

[21] E. A. Lee and D. G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75:1235–1245, September 1987.

[22] G. Manimaran and C. S. R. Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Trans. Parallel and Distributed Systems*, 9:1137?1152, November 1998.

[23] P. Marwedel. *Embedded system design*. Springer, 2010.

[24] P. Marwedel. Workshop on mapping of applications to MPSoCs. In *http://www.artist-embedded.org/artist/Program,2298.html*, 2011.

[25] T.G. Mattson, R.F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2010*, Nov. 2010.

[26] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *45th annual Design Automation Conference (DAC)*, pages 574–579, 2008.

[27] Devendra Rai, Hoeseok Yang, Iuliana Bacivarov, Jian-Jia Chen, and Lothar Thiele. Worst-case temperature analysis for real-time systems. DATE11, Grenoble, France, 2011.

[28] Tajana Simunic-Rosing, Ayse Kivilcim Coskun, and Keith Whisnant. Temperature aware task scheduling in MPSoCs. *Design, Automation and Test in Europe (DATE)*, pages 1659–1664, 2007.

[29] Kevin Skadron et al. Temperature-aware microarchitecture: modeling and implementation. *ACM T. Arch. and Code Opt.*, 1(1):94–125, 2004.

[30] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. *Proc. International Symposium on Computer Architecture (ISCA)*, pages 276–287, 2004.

[31] J. Teich. Invasive algorithms and architectures. *it - Information Technology*, 50(5):300–310, 2008.

[32] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive Computing: An Overview. In *Multiprocessor system-on-chip: hardware design and tool integration*, pages 241–268. Springer, 2011.

[33] Lothar Thiele et al. Mapping applications to tiled multiprocessor embedded systems. In *Proc. ACSD*, pages 29–40, 2007.

[34] Lothar Thiele, Lars Schor, Hoeseok Yang, and Iuliana Bacivarov. Thermal-aware system analysis and software synthesis for embedded multi-processors. In *Proc. Design Automation Conference (DAC)*, San Diego, California, USA, 2011. ACM.

[35] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.

[36] D. Wentzlaff, P. Griffin, H. Hoffmann, Liewei Bao, B. Edwards, C. Ramey, M. Mattina, Chyi-Chang Miao, J.F. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15 –31, Sept.-Oct. 2007.