Real-Time Task Migration for Dynamic Resource Management in Many-Core Systems

Behnaz Pourmohseni
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
behnaz.pourmohseni@fau.de

Fedor Smirnov
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
fedor.smirnov@fau.de

Stefan Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
stefan.wildermann@fau.de

Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
juergen.teich@fau.de

Abstract

Dynamic resource management strategies in embedded many-core systems rely on task migration to adapt the deployment (mapping) of applications dynamically, e.g., for thermal/power management or load balancing. In case of hard real-time applications, however, the current practice of on-line application adaptation is limited to reconfiguring the whole application between a set of statically computed mappings with statically verified timing guarantees. This heavily restricts the application's adaptability. To enable hard real-time task migrations in many-core systems without relying on a static analysis, this paper presents (i) a *predictable task migration mechanism* supported with (ii) a lightweight *migration timing analysis* and (iii) a lightweight *migration timing feasibility check* which can be applied on-line to bound on the worst-case temporal overhead of a migration and examine the admissibility of this overhead w.r.t. the hard real-time requirements of the application. For a variety of applications and many-core platforms, we experimentally demonstrate the feasibility of hard real-time task migrations, the lightness of the proposed timing analysis and feasibility check for on-line use, and the advantage of the proposed task migration approach over mapping reconfiguration as the state-of-the-art real-time adaptation approach for many-core systems.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Multicore architectures

Keywords and phrases Hard real-time, task migration, timing analysis, dynamic resource management, multi-core, many-core

Digital Object Identifier 10.4230/OASIcs.NG-RES.2020.5

Funding This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 - TRR 89 Invasive Computing.

1 Introduction

The ever-increasing number of applications hosted on a shared multi/many-core platform in modern embedded systems engenders a highly dynamic environment: Different applications are launched and terminated on demand and independently from each other, running applications are exposed to workload variation and fluctuating performance requirements, and platform resources may become unavailable unexpectedly, e.g., due to the emergence of

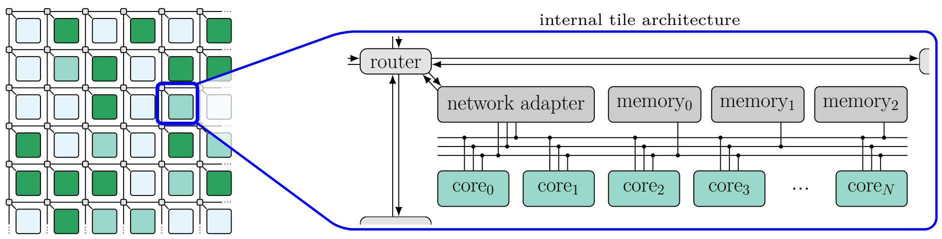


Figure 1 A heterogeneous tiled many-core architecture. Tiles are interconnected by a NoC. Each tile comprises a set of cores, a set of memories, and a network adapter, interconnected via buses.

thermal hot spots or hardware faults. Such events are typically addressed using dynamic resource management strategies which adapt the deployment of running applications. These strategies chiefly rely on *task migration* for rearranging the applications.

Migration-based resource management strategies can be viewed as an ensemble of two components: a migration policy and a migration mechanism. The *migration policy* determines *which* task(s) must be migrated *when* and *whereto*. A major factor taken into account during this selection process is the overhead associated with each migration option, e.g., the latency or the resource requirement of the migration process. These overheads are primarily a byproduct of the underlying *migration mechanism* which determines *how* a migration is performed. The choice of migration mechanism, in turn, depends on the target hardware architecture, particularly, its interconnection scheme and memory organization.

Many-core platforms, e.g. [8, 21, 36], are typically organized as a set of tiles with a Network-on-Chip (NoC) interconnection and a distributed No Remote Memory Access (NORMA) storage scheme for scalability [26], see, e.g., Fig. 1. Each tile comprises a set of cores, a set of memories, and a Network Adapter (NA), interconnected via a set of memory buses. This infrastructure enables the transmission of messages both between cores located on the same tile (*intra-tile* transmission) and between cores located on different tiles (*inter-tile* transmission). In the context of task migration, intra-tile task migrations are realized through the on-tile memories, oftentimes implicitly. The distributed memory scheme between tiles, however, necessitates inter-tile task migrations to be realized by explicit relocation of the task context between the source and destination tiles over the NoC.

Motivation. Existing works in the area of real-time task migration are either tailored to *soft* real-time constraints and try to reduce the number of deadline misses [1, 6], or assume a universal shared-memory scheme which, in the context of many-core systems, restricts their scope of applicability to intra-tile migrations only [19, 38]. Recently, *composable* many-core systems have emerged, primarily to cope with the immense systems dynamism and design complexity [2, 17, 41]. In a composable many-core system, e.g. [17], running applications are decoupled from each other using explicit reservation of resources (or resource budgets) required by each application so as to establish a spatial and/or temporal isolation between concurrent applications [2, 23]. This enables the worst-case temporal behavior of each application to be analyzed based on its reserved resources (or resource budgets), irrespective of the choice and behavior of the other applications that may run concurrently.

Contribution. In this paper, we exploit system composability to enable hard real-time task migrations without relying on a static timing analysis and verification. To that end, we present (i) a *predictable migration mechanism* which complies with the storage- and communication schemes of many-core systems and can be employed for both intra- and inter-tile migrations, even in the case of migrations between cores of different types. We supply the proposed migration mechanism with (ii) a lightweight *migration timing analysis* which can be used on-line to calculate a safe bound on the worst-case latency of each migration process. To verify the real-time conformity of a migration, we then present (iii) a lightweight *migration timing feasibility check* which examines the admissibility of the migration latency w.r.t. the given hard real-time deadline of the application and the changes in its timing behavior during and after the migration. Our experimental results demonstrate the feasibility of hard real-time task migrations, the lightness of the proposed timing analysis and feasibility check for on-line use, and the advantage of the proposed task migration approach over the state-of-the-art hard real-time adaptation approach, namely, mapping reconfiguration.

2 Related Work

A large body of work exists on task migration in multi/many-core systems used for load balancing [4, 14, 22], temperature balancing [16, 24, 27], or fault resilience [3, 37]. They, however, either (i) rely on assumptions about the platform which do not necessarily apply to embedded many-core platforms, or (ii) disregard the temporal overhead of migration, making them inapplicable for hard real-time applications. For instance, in [1, 5, 19, 20, 30, 37], a globally shared-memory scheme is assumed for context migration while many-core systems typically manifest a distributed NORMA scheme [26]. Likewise, the migration approaches in [1, 12, 15, 27, 30] rely on a full/partial static replication of tasks on every memory in the system, which imposes an immense storage overhead that is often not tolerable in embedded many-core systems. From a predictability viewpoint, only a few existing migration approaches investigate the timing overhead of task migration [1, 6, 19, 38]. They, however, either assume soft real-time requirements and do not provide timing *guarantees* [1, 6] or investigate hard real-time task migration but rely on assumptions such as a globally shared-memory scheme which makes them inapplicable for inter-tile migrations in many-core systems [19, 38].

In the context of dynamic many-core systems, existing approaches [11, 32, 33, 40] for hard real-time application adaptation verify the admissibility of migration overhead using compute-intensive *static* timing analyses. Authors in [11] investigate real-time *system reconfigurations* between statically known system modes, each corresponding to a unique choice and deployment of active applications. Since the number of system modes and migrations per mode transition grows exponentially with the number of applications, this approach is generally not considered a viable solution for highly dynamic systems. To improve scalability, authors in [32, 33, 40] investigate per-application composable *mapping reconfigurations* in which each running application can be independently reconfigured between a set of *statically* computed mappings without affecting the other running applications.

In this paper, we present a task migration mechanism and timing analysis which, compared to mapping reconfiguration, enables a finer adaptation granularity as it empowers the real-time migration of *any subset* of an application's tasks *without* relying on a static analysis. Contrarily to existing migration solutions, our approach complies with the distributed memory scheme of embedded many-core systems. It is supported with a lightweight timing analysis and feasibility check which bound the worst-case temporal overhead of the migration processes at run time and examine the admissibility of this overhead w.r.t. the application deadline and the changes in its timing behavior *during* and *after* the migrations.

3 System Model

3.1 Platform Architecture

The target many-core platform is assumed to be organized as a set of (possibly heterogeneous) tiles interconnected by a Network-on-Chip (NoC), see, e.g., Fig. 1. Each tile comprises a set of homogeneous cores, memories, and a Network Adapter (NA), interconnected via buses.

Composability. The platform is assumed to be devoid of timing anomalies [35] and fully composable [2, 17], so that applications can share resources without affecting each other's worst-case timing behavior. Composability is established by means of exclusive reservation of resources (or reservation of periodic time budgets on resources) per application at its launch time. To establish this scheme, each potentially shared resource, i.e., core, bus, NoC link, and NA, must have a contentionless time-triggered arbitration policy, e.g., Time-Division Multiplexing (TDM) or Weighted Round-Robin (WRR). In this context, the worst-case timing behavior of each application can be analyzed based on its required resources (or resource budgets). As a result, as long as the reserved resource budgets of an application remain intact, its analyzed worst-case timing guarantees will hold, regardless of the presence and the behavior of other applications which utilize the remaining budget of these resources.

Memory Model. We consider a distributed NORMA scheme between tiles which is common for many-core systems [26]. Under this memory scheme, inter-tile data exchanges are realized by means of explicit message passing between communicating tiles over the NoC, while intra-tile data exchanges are realized through dedicated spaces in the memories on the respective tile. To achieve storage composability, the memory space in each tile is dynamically partitioned among tasks executed on it and messages produced and/or consumed on it.

NoC Model. The NoC is assumed to have a wormhole-switched- [29] and credit-based virtual-channel [7] flow control, see, e.g., the NoC in [18]. Under wormhole switching, packets are decomposed into so-called flits which are routed independently from each other in pipeline. Virtual channels provide multiple buffers per link which enables transmission preemption and composable link sharing among multiple communication flows. For each flow, the required bandwidth budget can be reserved on each link located on its transmission route, and its transfer latency can be analyzed based on its reserved budget, irrespective of the other flows.

3.2 Application and Mapping

We consider data-flow applications with a hard real-time constraint on their end-to-end latency (makespan), denoted as the application deadline. Each application is specified by an acyclic task graph (DAG) $G_P(T \cup M, E)$ where T denotes the set of tasks and M denotes the set of unicast messages, each exchanged between one pair of tasks. E is a set of directed edges which represent data dependencies among tasks and messages. For each task $t \in T$, the Worst-Case Execution Time (WCET) C_t per core type, the minimum interarrival time P_t, and the maximum context size B_t are given. For each message $m \in M$, the minimum interarrival time P_m and the maximum payload size B_m are given.

To execute an application, a so-called *mapping* of it on the platform is used which specifies (i) the binding and budget of the tasks on cores and (ii) the routing and budget of the inter-tile messages on the NoC. The Worst-Case Response Time (WCRT) L_t of each task $t \in T$ and the Worst-Case Traversal Time (WCTT) L_m of each message $m \in M$ are derived based on the budget reserved for each task (message) on its bound core (NoC route). For this purpose, we use the timing analysis from [31].

4 Real-Time Task Migration

Resource management in many-core systems, particularly, the migration of tasks, is typically controlled and operated by a so-called Run-time Manager (RM), see [39] for an overview. In the following, we consider a scenario where, during the execution of an application, task migration becomes necessary to address a run-time event, e.g., a thermal hot spot. Assume that the RM has selected a subset of the application's tasks for migration to different destinations. Before starting the migrations, the RM must first check the availability of resources required by each migrating task. These are (i) the target core, (ii) the post-migration NoC routes for inter-tile messages to/from the migrating task, and (iii) migration routes for data transfer between the source and destination tiles in case of inter-tile migrations.

In a non-real-time context, the RM performs the migrations after the availability of the required resources for all migrating tasks is verified. In a hard real-time context, however, the migrations can take place only after the RM also verifies that (iv) the timing overhead imposed during the migrations and (v) the changes in the timing behavior of the application after the migrations cannot lead to a violation of its real-time deadline. To enable this verification, in Section 4.1 we present a migration mechanism that enables the RM to migrate tasks in a predictable fashion and transparently to the application. In Section 4.2, we present a migration timing analysis which enables the RM to bound the worst-case latency of the steps involved in the migration of each task and, then, the end-to-end latency of the multi-task migration process. In Section 4.3, we present a migration timing feasibility check which enables the RM to verify the real-time conformity of the migrations w.r.t. the end-to-end migration latency and the changes in the timing behavior of the application during and after the migrations. Finally, we present an illustrative example in Section 4.4 and elaborate on the run-time overhead and complexity of our approach in Section 4.5.

4.1 Migration Mechanism

This section presents a task migration mechanism which enables the RM to perform task migrations in a predictable manner and transparently to the application. Our migration mechanism is non-preemptive. This enables migrations between heterogeneous cores using *fat binaries* without requiring source code modification and state transformation mechanisms which are typically not available in embedded systems. A fat binary comprises a set of binaries, one per Instruction Set Architecture (ISA), from which the fitting binary is selected at the migration destination, see [28]. We distinguish between intra- and inter-tile migrations:

Intra-Tile Task Migration. If a task is to be migrated between two cores on the same tile, the migration is realized implicitly via the memories on the tile. Here, the RM simply schedules the task for its next execution iteration (job) on the target core instead of the source core. The latency of this process can be safely bounded by the (known) worst-case context-switch latency L_{OS} of the operating system.

Inter-Tile Task Migration. Migrating a task between different tiles requires an explicit transfer of the task's dataset between the source and destination tiles. To that end, first the execution of the migrating task is suspended non-preemptively, i.e., after completing its current job. At the same time, its input/output (i/o) messages are suspended by blocking the injection of new messages into the NoC while allowing the already-injected messages to reach their destination node. The former ensures execution consistency between the jobs executed before the migration and the jobs executed after the migration, while the latter is crucial to prevent communication inconsistencies that may arise, e.g., due to out-of-order delivery or even loss of input messages if they arrive at the old location *after* the migration process. Note that system services such as message forwarding or buffer reordering for resolving these issues are not typical for embedded systems. After the current job is completed and the i/o messages are suspended, the relocation process between the migration source- and destination tiles begins. In this step, the task's context, its unprocessed input messages, and its blocked output messages – all residing in the source tile's memory – are relocated to the destination tile. The task's execution is resumed after the relocation process has completed.

4.2 Migration Timing Analysis

This section presents a migration timing analysis that enables the RM to bound the worst-case end-to-end latency of migration processes. To that end, let $\hat{T} \subseteq T$ denote the set of tasks selected for inter-tile migration. Also, let function $M_{\text{io}}(t)$ provide the set of input and output messages of task $t \in T$. For each task $t \in \hat{T}$, the worst-case migration latency consists of two components: (i) suspension latency $\delta_{\text{susp}}(t)$ and (ii) relocation latency $\delta_{\text{reloc}}(t)$. In the following, we present a lightweight timing analysis to bound the suspension- and relocation latency of each migrating task, and, subsequently, the end-to-end latency of the multi-task migration process for the two predominant cases of sequential and parallel migrations.

4.2.1 Suspension Latency

The suspension process of a migrating task $t \in \hat{T}$ – which begins after the current job of t has completed – involves two parallel operations: (i) storing the state of t in the tile memory and (ii) suspending the i/o messages of t. *State storage* is performed by the operating system. The latency of this process is bounded by the (known) worst-case context-switch latency L_{OS} of the operating system. *Communication suspension* is realized by blocking the injection of new input messages and output messages of the migrating task into the NoC and allowing the already-injected i/o messages to reach their destination. In the worst case, the suspension process is initiated right after the i/o messages are injected into in the NoC. Since each message m is guaranteed to be transmitted within its WCTT L_m, the worst-case latency for suspending all i/o messages of t can be bounded by the largest WCTT among its i/o messages. Taking into account the two parallel operations above, (i) and (ii), the worst-case suspension latency $\delta_{\text{susp}}(t)$ of each migrating task $t \in \hat{T}$ can be bounded as:

$$\delta_{\text{susp}}(t) = \max\left\{L_{OS}, \max_{m \in M_{\text{io}}(t)} \{L_m\}\right\} \tag{1}$$

4.2.2 Relocation Latency

The relocation of a migrating task $t \in \hat{T}$ begins only after t is suspended and involves the transfer of the *migration dataset* of t from the memory on the source tile to the destination tile. The migration dataset denotes the data required for a seamless resumption of t's execution at the destination tile. It contains t's context (code, state, etc.) of size B_t and its unprocessed input- and blocked output messages $m \in M_{\text{io}}(t)$, residing in the source tile's memory. Thus, the size of the migration dataset for task $t \in \hat{T}$ is bounded by $B_{\text{mig}}(t) = B_t + \sum_{m \in M_{\text{io}}(t)} B_m$, where B_m denotes the maximum payload size of message m.

The migration dataset is transferred to the destination tile in three steps: (i) the NA on the source tile reads the dataset from the memory, decomposes it into flits, and injects the flits into the NoC. (ii) The flits are then transferred over the NoC to the destination tile.

Finally, (iii) the NA on the destination tile reconstructs the dataset from the flits and stores it in the memory. The worst-case relocation latency $\delta_{\mathrm{reloc}}(t)$ of a task $t \in \hat{T}$ can be bounded using Eq. (2). Here, the first term bounds the latency of steps (i) and (iii), which we derive using the NA latency analysis from [31]. Note that the source and destination NAs have identical worst-case latencies, as they read/write the same amount of data $B_{\mathrm{mig}}(t)$ from/to the memories. The second term in Eq. (2) bounds the NoC latency for transferring $B_{\mathrm{mig}}(t)$ over the migration route $\rho_{\mathrm{mig}}(t)$, which we derive using the NoC latency analysis from [33]. Both NA- and NoC analyses [31, 33] are lightweight and can be used on-line.

$$\delta_{\mathrm{reloc}}(t) = 2 \times L_{\mathrm{NA}}\Big(B_{\mathrm{mig}}(t)\Big) + L_{\mathrm{NoC}}\Big(B_{\mathrm{mig}}(t), \rho_{\mathrm{mig}}(t)\Big) \tag{2}$$

4.2.3 End-To-End Migration Latency

The end-to-end migration latency denotes the overall time overhead imposed on the regular execution of the application due to the migration of one or more tasks. It reflects the interval between the moment when the state storage of the first migrating task begins and the moment when the relocation processes for all migrating tasks are completed. In case of a single-task migration, the end-to-end migration latency is bounded by the sum of the suspension time $\delta_{\mathrm{susp}}(t)$ and the relocation time $\delta_{\mathrm{reloc}}(t)$ of that task t. If multiple tasks are to be migrated, the migrations may be performed (i) in parallel or (ii) sequentially. These two approaches enable the RM to draw a trade-off between the end-to-end migration latency and the amount of NoC budget that must be reserved for establishing the migration routes.

Parallel Migrations. In case of parallel migrations, for each migrating task t, a suspension latency $\delta_{\mathrm{susp}}(t)$ and a relocation latency $\delta_{\mathrm{reloc}}(t)$ is imposed. Thus, the end-to-end latency of parallel migrations can be bounded using Eq. (3). Note that parallel migrations are possible only if sufficient budget on NoC links is available so that the RM can reserve a migration route $\rho_{\mathrm{mig}}(t)$ for each migrating task $t \in \hat{T}$. Congestion could then particularly occur when multiple migrating tasks have overlapping migration routes.

$$\delta_{\mathrm{mig}}^{\mathrm{par}}(\hat{T}) = \max_{t \in \hat{T}} \Big\{ \delta_{\mathrm{susp}}(t) + \delta_{\mathrm{reloc}}(t) \Big\} \tag{3}$$

Sequential Migrations. In case of a sequential relocation of tasks, the end-to-end migration latency depends on the order in which the migrating tasks are relocated. Here, it may happen that the suspension of those tasks that are decided to be migrated first takes longer than the suspension of those that are decided to be migrated after the former. As a result, the latter suffer an idle time before the relocation of the former begins. Here, the worst-case scenario arises when (i) the task $t' \in \hat{T}$ chosen to be migrated first is the one with the highest WCRT, i.e., $L_{t'} = \max_{t \in \hat{T}}\{L_t\}$, (ii) the suspension request is issued right after t' starts its execution iteration, and (iii) at least one other migrating task $\tilde{t} \in \hat{T}$ has finished its execution iteration and updated its state in the memory prior to the suspension request. In this situation, $\tilde{t}$ undergoes the highest possible idle time before the relocation of the first migrating task t' begins. This idle time is guaranteed not to exceed the sum of t''s WCRT $L_{t'}$ and worst-case suspension time $\delta_{\mathrm{susp}}(t')$. Thus, the worst-case migration latency for a sequential relocation of migrating tasks can be bounded as:

$$\delta_{\mathrm{mig}}^{\mathrm{seq}}(\hat{T}) = \max_{t \in \hat{T}} \Big\{ L_t + \delta_{\mathrm{susp}}(t) \Big\} + \sum_{t \in \hat{T}} \delta_{\mathrm{reloc}}(t) \tag{4}$$

4.3 Migration Timing Feasibility Check

Task migration affects the temporal behavior of the application twofold: First, the regular execution of migrating tasks and the injection of their i/o messages into the NoC are suspended during the migration process. Second, the WCRT of each migrating task and the WCTT of its i/o messages may change after the migration; The WCRT of a task may change, e.g, if its pre- and post-migration cores are heterogeneous. The WCTT of a message may change, e.g., if its pre- and post-migration NoC routes have different lengths.

We present a lightweight migration timing feasibility check to enable the RM to examine whether performing a given set of migrations can lead to the violation of the application's deadline, taking into account the worst-case migration latency as calculated in Section 4.2 and the changing timing behavior of the application during and after the migrations. The migration timing feasibility check must verify that the application deadline will be respected by the end-to-end latency of each application input which either (i) arrives before the migrations and is processed by some migrating tasks before their migration and by some others after their migration or (ii) arrives during/after the migrations and is, therefore, processed by the migrating tasks after their migration. We examine the satisfaction of the given application deadline for both of these cases simultaneously by calculating a safe upper bound on the end-to-end latency of any application input as follows:

i For each migrating task $t \in \hat{T}$, the post-migration WCRT L'_t is calculated using the response time analysis from [31]. For all other tasks $t \in T \backslash \hat{T}$, we consider $L'_t = L_t$.

ii For each message $m \in M$ to/from the migrating tasks, the post-migration WCTT L'_m is calculated using the traversal time analysis from [31]. For other messages, $L'_m = L_m$.

iii For each application task/message $x \in T \cup M$, a safe bound on x's pre- and post-migration latency is derived as $\hat{L}_x = \max\{L_x, L'_x\}$, referred to as the *compound latency* of x.

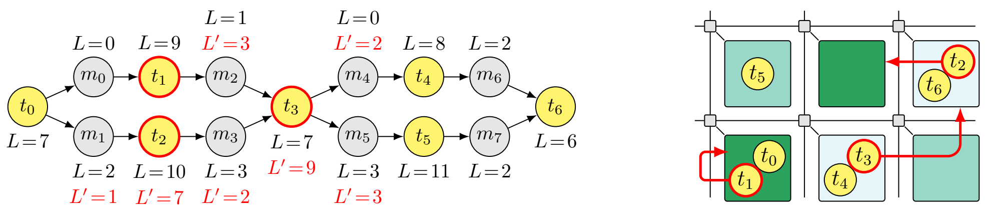
iv The latency of the longest path in the application DAG is derived using the DFS algorithm [13] where the compound latency of each task/message is used as its weight. The result is referred to as the *compound application latency* and denoted by $\hat{L}_{\text{app}}$.

The compound application latency $\hat{L}_{\text{app}}$ provides a safe bound on the end-to-end latency of any application input whose processing may be affected by the migrations in question. Therefore, the RM can check the real-time conformity of the migrations by verifying that $\hat{L}_{\text{app}} + \delta_{\text{mig}}(\hat{T})$ does not exceed the given application deadline. Here, $\hat{L}_{\text{app}}$ bounds the end-to-end latency of the application and $\delta_{\text{mig}}(\hat{T})$ (derived in Section 4.2) bounds the end-to-end latency of the migrations.

4.4 Example

Consider the exemplary application depicted in Fig. 2a which is mapped on four tiles of a many-core architecture as shown in Fig. 2b. The application tasks t_0–t_6 communicate with each other via messages m_0–m_7. For brevity, the NoC routes of messages and the internal layout of tiles (including the binding of tasks to cores, the memories, and the NAs) are not depicted in Fig. 2b. The WCRT L_t of each task t and the WCTT L_m of each message m are also given in Fig. 2a. Assume a scenario where the RM has selected tasks t_1–t_3 for migration to the destinations indicated by red arrows in Fig. 2b. Task t_1 is selected for intra-tile migration, whereas tasks t_2 and t_3 are selected for inter-tile migration, thus, $\hat{T} = \{t_2, t_3\}$.

To check whether the migration of t_2 and t_3 can lead to the violation of the application's deadline, the RM first calculates the end-to-end latency of the migrations. Assuming a context-switch latency of $L_{OS} = 1$, Eq. (1) bounds the suspension latency of the migrating tasks as $\delta_{\text{susp}}(t_2) = \max\{1, \max\{2, 3\}\} = 3$ and $\delta_{\text{susp}}(t_3) = \max\{1, \max\{1, 3, 0, 3\}\} = 3$.



(a) example application. **(b)** on-chip deployment (mapping).

Figure 2 (a) Example application annotated with pre-/post-migration latencies of tasks and messages and (b) its pre-migration mapping on the chip, used in the illustrative example in Section 4.4.

Then, assuming relocation latencies of $\delta_{\text{reloc}}(t_2) = 4$ and $\delta_{\text{reloc}}(t_3) = 6$, the end-to-end migration latency of t_2 and t_3 is guaranteed not to exceed $\delta^{\text{par}}_{\text{mig}}(\hat{T}) = \max\{(3+4),(3+6)\} = 9$ in case of parallel migrations, or $\delta^{\text{seq}}_{\text{mig}}(\hat{T}) = \max\{(10+3),(7+3)\} + (4+6) = 23$ in case of sequential migrations, derived using Eq. (3) and Eq. (4), respectively.

For migration timing feasibility check, assume that the RM has derived – using the analysis from [31] – the post-migration WCRT L'_t of each migrating task $t \in \hat{T}$ and the post-migration WCTT L'_m of t's i/o messages $m \in M_{\text{io}}(t)$ as given in Fig. 2a. Based on these, the compound application latency is bounded to $\hat{L}_{\text{app}} = 53$, following steps (i)–(iv) in Section 4.3. Recall that $\hat{L}_{\text{app}} = 53$ bounds the end-to-end latency of application inputs that are affected by the migration process. In our example, this is the latency for an input that passes through t_0, m_1, t_2, and m_3 before the migrations, is blocked at the input buffer of t_3 prior to the migration process, is relocated with t_3 during the migrations, and passes through t_3, m_5, t_5, m_7, and t_6 after the migrations. Based on the latency bounds above, the RM performs the migrations only if the application deadline is at least $\delta^{\text{par}}_{\text{mig}}(\hat{T}) + \hat{L}_{\text{app}} = 9+53 = 62$ in case of parallel migrations, or $\delta^{\text{seq}}_{\text{mig}}(\hat{T}) + \hat{L}_{\text{app}} = 23+53 = 76$ in case of sequential migrations.

4.5 Run-Time Overhead and Complexity

Any analysis targeted for on-line use must be lightweight so as to introduce an acceptable overhead for the RM. In the following, we elaborate on the computational complexity of the proposed migration timing analysis and feasibility check. Note that the WCRT and WCTT analyses adopted from [31], and the NA- and NoC latency analyses adopted from [31] and [33], respectively, are constant-time non-iterative operations with a complexity of $O(1)$.

The migration timing analysis presented in Section 4.2 embodies a 2-level nested loop where the outer loop iterates through migrating tasks and the inner loop iterates through their i/o messages. Since each message is unicast (has one producer and one consumer, see Section 3.2), the inner loop can have a maximum total of $2\,|M|$ iterations, resulting in a linear time complexity of $O(|T| + 2\,|M|) = O(|T| + |M|)$ for the migration timing analysis.

For the migration timing feasibility check, the main compute overhead stems from the calculation of the compound application latency in steps (i)–(iv) in Section 4.3. Here, steps (i)–(iii) are implemented by simple loops with a computational complexity of $O(|T|)$, $O(|M|)$, and $O(|T| + |M|)$, respectively. Having the application DAG provided as adjacency lists, the DFS algorithm in step (iv) will have a complexity of $O(|T| + |M|)$. Therefore, the migration timing feasibility check presented in Section 4.3 has a linear time complexity of $O(|T| + |M|)$. When examining the real-time conformity of a (possibly multi-task) migration, the RM applies the migration timing analysis and the feasibility check in succession. This introduces a compute overhead of linear time complexity $O(|T| + |M|)$ for the RM, rendering the proposed migration timing analysis and feasibility check scalable for on-line use.

5 Experimental Results

For our experiments, we consider two heterogeneous tiled many-core architectures with 6×6 and 8×8 tiles, respectively. Each tile is composed of four homogeneous cores while each platform comprises tiles of three different core types. Every shared resource (core, bus, NA, and NoC link) has a WRR arbitration policy. For the NoC, the XY-routing algorithm [29] is used. We consider four hard real-time applications from areas of automotive (18 tasks, 21 messages), telecommunication (14 tasks, 20 messages), consumer (11 tasks, 12 messages), and networking (7 tasks, 9 messages) provided by the Embedded System Synthesis Benchmarks Suite (E3S) [10]. To obtain a set of mappings for each application per architecture, we use the OpenDSE framework [34] to perform a Design Space Exploration (DSE), employing the NSGA-II evolutionary algorithm [9] provided by the Opt4J optimization framework [25]. The DSE is performed over $1,000$ generations and retains a population of 100 mappings. It optimizes the mappings w.r.t. five design objectives to be minimized: (i) distance to the hard real-time application deadline (set to 80% of the aggregate interarriavl time of tasks and messages on the longest path) evaluated using the analysis from [31], (ii) energy consumption evaluated based on [10] for cores and [42] for buses/NoC links with wire lengths of 5 mm and 2 mm, respectively, and (iii)–(v) number of allocated cores from each of the three core types. The DSE provides a set of Pareto-optimal mappings V_i per application i.

In our experiments, we investigate the feasibility and the effectiveness of the proposed real-time task migration approach in a case study on adaptive thermal management of many-core systems. Consider the scenario in which a real-time application i is launched using one of its precomputed mappings $v \in V_i$. During the execution of the application, the RM identifies the emergence of a thermal hot spot around one of the cores in use by the application which, consequently, necessitates the evacuation of the thermally affected core while guaranteeing that the evacuation process will not lead to the violation of the application's deadline. For the evacuation, we consider two adaptation approaches:

(i) *Mapping Reconfiguration.* In this approach, the RM reconfigures the application to another one of its precomputed mappings which does not depend on the thermally affected core. To that end, the RM iterates through the mappings $v' \in V_i \setminus \{v\}$ and checks per mapping (i) the availability of its required cores and NoC routes, (ii) the availability of migration routes for the relocation of (potentially all) tasks, and (iii) the real-time conformity of the reconfiguration process. We implement this approach using the mapping reconfiguration mechanism and timing analysis from [33] which are developed based on a sequential migration of tasks. This approach represents the state of the art in hard real-time application adaptation. Here, the evacuation of the thermally affected core is considered successful iff a mapping is found which passes both the resource checks, (i) and (ii), and the timing check, (iii).

(ii) *Task Migration.* In this approach, the RM migrates only those tasks that are running on the thermally affected core. We implement this approach using the proposed migration mechanism, supported by our migration timing analysis and timing feasibility check for the worst-case timing verification of the migrations. For the sake of comparability with mapping reconfiguration, the migrations are performed sequentially. For a migration-based evacuation, the RM iterates through the platform tiles (excluding the heated tile) and checks for each candidate tile, (i) the availability of a free core, (ii) the availability of NoC routes for i/o messages of the migrating tasks after the migration, and (iii) the availability of a NoC route for the relocation of migrating tasks. If the availability of all required resources is verified,

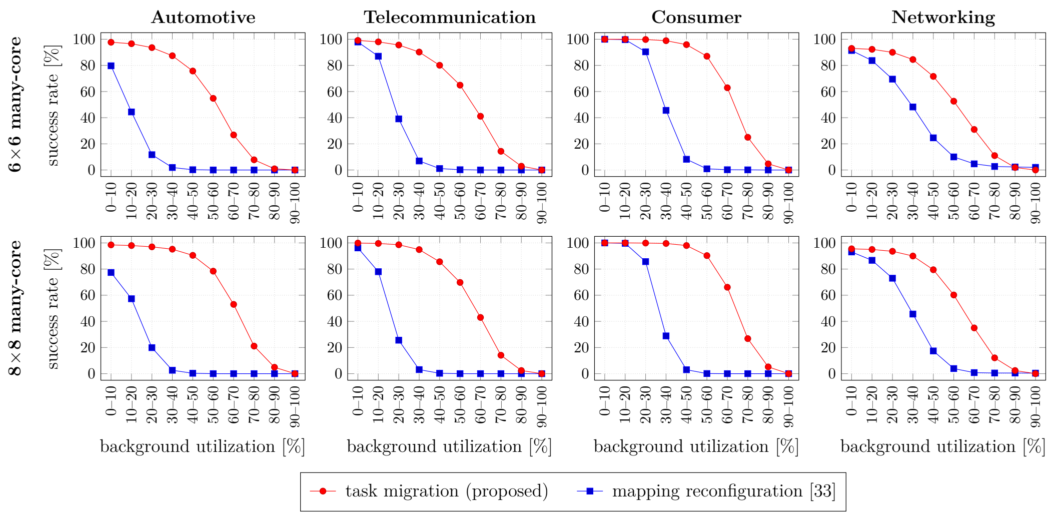


Figure 3 Success rate of task migration and mapping reconfiguration at different background utilization levels. The plots in each column (or row) correspond to one application (or architecture).

the RM performs (iv) the migration timing analysis and feasibility check. If the required resources are not available or the timing check is not passed, the RM continues its search through the remaining tiles. The evacuation is considered successful iff a destination tile is found with passes both the resource checks, (i)–(iii), and the timing check, (iv).

We perform the evacuation experiment for each mapping $v \in V_i$ of each application i as follows: First, application i is launched on an empty platform using mapping v. Then, we introduce additional (background) load into the system by iteratively occupying free resources (cores and NoC links) at random, thereby, generating different *background utilization levels*. At each utilization level, we then iterate through the cores in use by the application and, in each iteration, mark one core as an emerging hot spot so that its evacuation becomes necessary in near future. Then, for each investigated approach, i.e., mapping reconfiguration and task migration, we check whether the affected core can be evacuated successfully.

Evacuation Success. For each background utilization level, we record the evacuation success of each approach. Figure 3 illustrates the success rate of the two approaches versus background utilization level per application (plot column) on each architecture (plot row). The reported results are an average over five runs of DSE per application and architecture and 20 repetitions of the run-time thermal management experiment per DSE to incorporate diverse mixes of preoccupied resources for each background utilization level. The obtained results offer two major insights: First, the high success rate of task migration demonstrates the practicality of task migration also in a hard real-time context. Second, compared to mapping reconfiguration, task migration offers a substantially higher success rate, demonstrating its advantage over mapping reconfiguration as a real-time deployment adaptation approach. Among all applications and architectures, task migration exhibits an up to 95% higher success rate (35% on average), compared to mapping reconfiguration. This success difference roots in three advantages of task migration over mapping reconfiguration: Since it often involves the relocation of only a subset of the application's tasks, task migration (i) requires a smaller set of resources which increases its chances of passing the resource checks, (ii) imposes a lower timing overhead which increases its chances of passing the timing check, and, thanks to its lightweight timing analysis and feasibility check, (iii) enables the RM to consider all possible adaptation options instead of a restricted set of statically computed options.

Run-Time Overhead. During the RM's search for a destination tile, the application continues its regular execution. Thus, the overhead of the search process is not critical w.r.t. the real-time constraints. However, to fit for on-line use, this overhead – which is mainly due to the resource- and timing checks – must be acceptable. In Section 4.5, we demonstrated the scalability of the proposed analyses which were shown to exhibit a linear time complexity of $O(|T| + |M|)$. To assess their overhead in absolute time, in the thermal management experiment, we also record the time spent during the RM's search process before the first destination is found which passes both the resource- and the timing checks – performed on an Intel i7-4770 CPU at 3.4 GHz with 32 GiB of RAM. The records denote an average overhead of 1.08 ms (standard deviation of 0.16 ms) for the resource checks and 0.57 ms (standard deviation of 0.06 ms) for the timing check. According to the results, the overhead of the proposed migration timing analysis and feasibility check is by an average of 47 % lower than that of the resource check which verifies their lightness of for on-line use.

6 Conclusion

In this paper, we proposed a predictable migration mechanism supported with a migration timing analysis and feasibility check to enable hard real-time task migrations in composable many-core systems. The proposed migration mechanism complies with the distributed memory scheme of many-core systems, and its supporting analysis is lightweight and, therefore, applicable for on-line use. Experimental results demonstrate the feasibility of hard real-time task migrations, the lightness of the proposed timing analysis and feasibility check for on-line use, and the advantage of the proposed task migration mechanism over mapping reconfiguration as the state-of-the-art hard real-time adaptation approach.

References

1 Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP journal on embedded systems*, 2008(1):518904, 2007.

2 Benny Akesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. Composability and predictability for independent application development, verification, and execution. In *Multiprocessor System-on-Chip*, pages 25–56. Springer, 2011.

3 Zaid Al-bayati, Brett H Meyer, and Haibo Zeng. Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 57–62, 2016.

4 Gabriel Marchesan Almeida, Sameer Varyani, Rémi Busseuil, Gilles Sassatelli, Pascal Benoit, Lionel Torres, Everton Alceu Carara, and Fernando Gehm Moraes. Evaluating the impact of task migration in multi-processor systems-on-chip. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design (SBCCI)*, pages 73–78, 2010.

5 Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 15–20, 2006.

6 Eduardo Wenzel Brião, Daniel Barcelos, Fabio Wronski, and Flávio Rech Wagner. Impact of task migration in NoC-based MPSoCs for soft real-time applications. In *Proceedings of the IFIP International Conference on Very Large Scale Integration*, pages 296–299, 2007.

7 William J Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 3(2):194–205, 1992.

8 Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey

Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013.

9 Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transaction on Evolutionary Computation (TEVC)*, 6(2):182–197, 2002.

10 Robert Dick. Embedded system synthesis benchmarks suite (E3S), 2010. URL: `http://ziyang.eecs.umich.edu/~dickrp/e3sdd/`.

11 Piotr Dziurzanski, Amit Kumar Singh, and Leandro Soares Indrusiak. Multi-criteria resource allocation in modal hard real-time systems. *EURASIP Journal on Embedded Systems*, 2017(1):30, 2017.

12 Ashaf El-Antably, Olivier Gruber, Frederic Rousseau, and Nicolas Fournel. Transparent and portable agent based task migration for data-flow applications on multi-tiled architectures. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 183–192, 2015.

13 Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.

14 Weiwei Fu, Tianzhou Chen, Chao Wang, and Li Liu. Optimizing memory access traffic via runtime thread migration for on-chip distributed memory systems. *The Journal of Supercomputing*, 69(3):1491–1516, 2014.

15 Laurent Gantel, Salah Layouni, Mohamed El Amine Benkhelifa, François Verdier, and Stéphanie Chauvet. Multiprocessor task migration implementation in a reconfigurable platform. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 362–367, 2009.

16 Yang Ge, Parth Malani, and Qinru Qiu. Distributed task migration for thermal management in many-core systems. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 579–584, 2010.

17 Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

18 Jan Heisswolf, Ralf König, Martin Kupper, and Jürgen Becker. Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Computers & Electrical Engineering*, 39(8):2603–2622, 2013.

19 Robert Hilbrich and J Reinier Van Kampenhout. Partitioning and task transfer on NoC-based many-core processors in the avionics domain. *Journal Softwaretechnik-Trends*, 30(3):6, 2011.

20 Simon Holmbacka, Wictor Lund, Sebastien Lafond, and Johan Lilius. Task Migration for Dynamic Power and Performance Characteristics on Many-Core Distributed Operating Systems. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 310–317, 2013.

21 Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 108–109, 2010.

22 Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: distributed resource management for on-chip many-core systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, 2011.

23 H Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2 edition, 2011.

24 Zao Liu, Sheldon X-D Tan, Xin Huang, and Hai Wang. Task migrations for distributed thermal management considering transient effects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, 23(2):397–401, 2015.

25 Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J: a modular framework for meta-heuristic optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1723–1730, 2011.

26 Guilherme Madalozzo, Liana Duenha, Rodolfo Azevedo, and Fernando G Moraes. Scalability evaluation in many-core systems due to the memory organization. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 396–399, 2016.

27 Fabrizio Mulas, David Atienza, Andrea Acquaviva, Salvatore Carta, Luca Benini, and Giovanni De Micheli. Thermal balancing policy for multiprocessor stream computing platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(12):1870–1882, 2009.

28 Peter Munk and Jan Richling. Migration-aware WCET estimation for heterogeneous multicores. *ACM SIGBED Review*, 11(3):22–25, 2014.

29 Lionel M Ni and Philip K McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

30 Michele Pittau, Andrea Alimonda, Salvatore Carta, and Andrea Acquaviva. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 59–64, 2007.

31 Behnaz Pourmohseni, Fedor Smirnov, Stefan Wildermann, and Jürgen Teich. Isolation-Aware Timing Analysis and Design Space Exploration for Predictable and Composable Many-Core Systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, pages 12:1–12:24, 2019.

32 Behnaz Pourmohseni, Stefan Wildermann, Michael Glaß, and Jürgen Teich. Predictable Run-Time Mapping Reconfiguration for Real-Time Applications on Many-Core Systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS)*, pages 148–157, 2017.

33 Behnaz Pourmohseni, Stefan Wildermann, Michael Glaß, and Jürgen Teich. Hard real-time application mapping reconfiguration for NoC-based many-core systems. *Real-Time Systems*, 55(2):433–469, 2019.

34 Felix Reimann, Martin Lukasiewycz, Michael Glaß, and Fedor Smirnov. OpenDSE – open design space exploration framework, 2018. URL: `http://opendse.sourceforge.net/`.

35 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

36 Tilera Corporation. Tile Processor Architecture Overview for the TILE-Gx Series, 2012.

37 Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. *ACM SIGBED Review*, 6(3):6, 2009.

38 Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Predictable Task Migration for Locked Caches in Multi-Core Systems. *ACM SIGPLAN Notices*, 46(5):131–140, 2011.

39 Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Design Automation Conference (DAC)*, pages 1–10, 2013.

40 Pranav Tendulkar and Sander Stuijk. A case study into predictable and composable MPSoC reconfiguration. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 293–300, 2013.

41 Andreas Weichslgartner, Stefan Wildermann, Michael Glaß, and Jürgen Teich. *Invasive Computing for mapping parallel programs to many-core architectures.* Springer, 2018.

42 Pascal T Wolkotte, Gerard JM Smit, Nikolay Kavaldjiev, Jens E Becker, and Jürgen Becker. Energy model of networks-on-chip and a bus. In *Proceedings of the International Symposium on System-on-Chip (SoC)*, pages 82–85, 2005.