TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings

Andrés Goens, Robert Khasanov,
Jeronimo Castrillon
Chair for Compiler Construction
TU Dresden,
Center for Advancing Electronics Dresden (cfaed)
{firstname.lastname}@tu-dresden.de

Marcus Hähnel, Till Smejkal,
Hermann Härtig
Chair of Operating Systems
TU Dresden,
Center for Advancing Electronics Dresden (cfaed)
{firstname.lastname}@tu-dresden.de

ABSTRACT

For embedded system software, it is common to use static mappings of tasks to cores. This becomes considerably more challenging in multi-application scenarios. In this paper, we propose TETRiS, a multi-application run-time system for static mappings for heterogeneous system-on-chip architectures. It leverages compile-time information to map and migrate tasks in a fashion that preserves the predictable performance of using static mappings, allowing the system to accommodate multiple applications. TETRiS runs on off-the-shelf embedded systems and is Linux-compatible. We embed our approach in a state-of-the-art compiler for multicore systems and evaluate the proposed run-time system in a modern heterogeneous platform using realistic benchmarks. We present two experiments whose execution time and energy consumptions are comparable to those obtained by the highly-optimized Linux scheduler CFS, and where execution time variance is reduced by a factor of 510, and energy consumption variance by a factor of 83.

CCS CONCEPTS

• **Software and its engineering** → **Run-Time environments**;
• **Computer systems organization** → *System on a chip*;

KEYWORDS

Heterogeneous, MPSoC, run-time, adaptivity, symmetry, multi-application

ACM Reference format:
Andrés Goens, Robert Khasanov,
Jeronimo Castrillon and Marcus Hähnel, Till Smejkal,
Hermann Härtig. 2017. TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings. In *Proceedings of SCOPES '17, Sankt Goar, Germany, June 12-14, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3078659.3078663

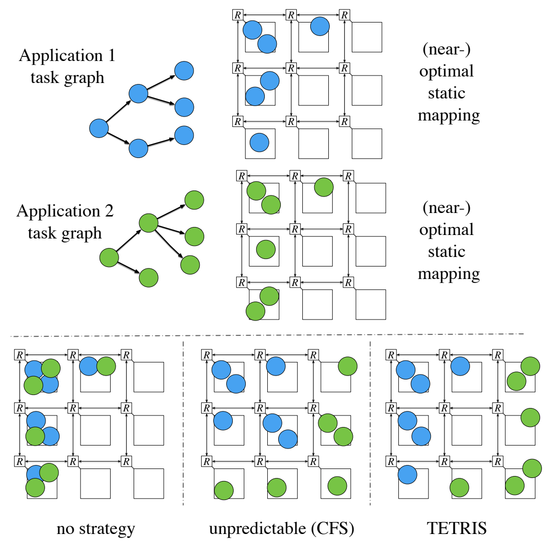


Figure 1: The basic idea behind TETRiS

1 INTRODUCTION

When designing software for embedded systems, reliability and predictability are usually just as much an asset as raw performance [1, 7, 16]. Today, with the increasing availability of high-performance embedded devices, like the ARM big.LITTLE™ platforms [11] or the many-core Epiphany-based systems [20], the lines between commodity, high-performance and embedded system devices and ecosystems are becoming increasingly blurred. Several modern embedded systems interact with the environment in a plethora of ways at the same time. This increases the unpredictability of software execution, since the workload of the system depends on many different and independent applications [16].

In several cases, the traditional notions of real-time software do not always apply anymore: many applications are too complex and variable, yet not critical enough for a rigorous worst-case execution-time analysis and verification. Consider a modern mobile phone. It has to keep several background processes running for the normal functioning of the phone, while being used for high-quality video playback. If then, a high-performance request gets scheduled, e.g., from a bluetooth device which wants to do simultaneous location and mapping using the phone, it would be ideal to keep the same performance for the other tasks, without changing the user experience. To ensure this, the allocation of resources might need to be adapted dynamically, while keeping the performance equal. We need approaches that ensure an execution that is predictable and efficient on average, even in the presence of dynamic changes in the system's load.

A large body of research has been dedicated to programming applications for these modern embedded systems [5, 8, 19, 21, 25]. Common approaches structure the application in an abstract way, and can thus make decisions based on these abstract models of computation. Usually: how to partition the application code, and where and how to schedule this partitioned code. The "where" question is sometimes also called *mapping*. In particular, it is a common practice to use static mapping and scheduling strategies at compile-time to obtain predictable outcomes [2].

However, flows considering static resource allocation and concentrating on a single application struggle when dealing with multiple applications. The estimated performance of an application does not consider other applications that will compete for system resources. This leads to significantly worse and less predictable performance when, at run-time, this isolated view of the application does not hold anymore. To avoid contention in the communication, hardware support like Network-on-Chip technologies or mechanisms at the hardware level to ensure isolation and predictable communication patterns between tasks [12, 15] are required. These concepts, along the lines with the concept of precision timed (PRET) machines [7], require very specific hardware, not all of which has been embraced by chip manufacturers yet.

In this paper we present a novel approach to keep applications in their isolated view at the software level, obtaining reliable performance in dynamic multi-application scenarios. While the methods should produce the best results on precision-timed machines, our implementation also works on off-the-shelf heterogeneous multiprocessor system-on-chips (MPSoCs) and is compatible with Linux. We call our approach the **T**ransitive **E**fficient **T**emplate **R**un-t**i**me **S**ystem (TETRiS).

Figure 1 summarizes the idea of the TETRiS approach. It uses near-optimal static mappings for the isolated applications, as depicted on the upper side of Figure 1. Then, it revolves around an innovative strategy for mapping multiple applications by identifying and selecting mappings with equivalent performance properties. By using a formal, mathematical model it can automatically and precisely identify, at compile-time, classes of mappings that will have the same performance on a system. Figure 1 shows what would happen when using no strategy at all, which results in a very inefficient resource distribution, an unpredictable redistribution, as would probably be computed, e.g., by the operating system's scheduler (CFS, the Completely Fair Scheduler, in the case of Linux), and TETRiS, where the structure of the mapping is preserved. Since our method leverages the structure of the architecture only, it is agnostic to the mapping strategy and its objectives. By using a strategic precomputation step, this can be done with almost no overhead at run-time.

We evaluate the TETRiS approach by embedding it into a state-of-the art commercial compiler for multicore systems, from the "SLX Tool Suite" by Silexica [23]. We use it then to execute multiple applications from the signal processing and multimedia domains on a modern heterogeneous architecture.

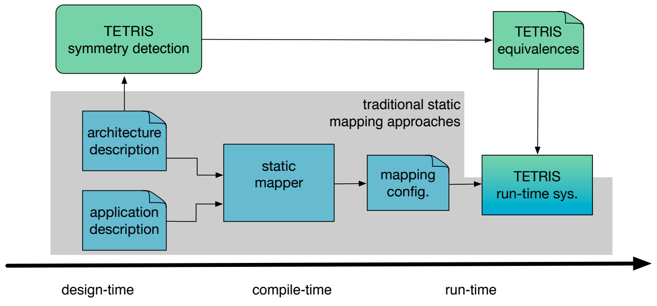


Figure 2: An overview of the TETRiS flow

The rest of the paper is structured as follows. Sections 2 and 3 describe the TETRiS system and the evaluation. Then, related work is discussed in Section 4, and finally, Section 5 concludes the paper.

2 THE TETRIS RUN-TIME SYSTEM

TETRiS, the **T**ransitive **E**fficient **T**emplate **R**un-t**i**me **S**ystem, is a run-time mapping and scheduling meta-strategy that enables the efficient use of near-optimal static-mappings in dynamic multi-application scenarios. The basics of the TETRiS runtime system are depicted in Figure 2. The gray area (blue boxes) represents the process flow of traditional static-mapping approaches, like [5, 8, 19, 21, 25]. On a high level of abstraction, it takes a model of the architecture and the application and uses them to calculate a near-optimal static mapping of the application to the architecture. The run-time system schedules the application respecting this mapping. The additions from TETRiS, the green boxes in Figure 2, calculate how to identify mappings that are equivalent using only the architecture description. This information is given to the TETRiS runtime system, which uses the original static mapping as a *template* to schedule applications by *transitively* iterating over the equivalence class of mappings.

In this section we describe the TETRiS system in detail. We integrate it into the multicore compiler provided by the SLX Tool Suite, which we briefly introduce. Since our approach is agnostic to the mapping strategy, any other static-mapping approach as described above could be used instead. TETRiS is not concerned by the properties of the mappings themselves, it just preserves them. Finally, we explain how the symmetry engine of TETRiS identifies and generates mapping equivalences, and how these are selected at load-time when a new application starts.

2.1 The SLX Tool Suite

The SLX Tool Suite is a set of tools for programming multicore systems. It is a commercial spin-off of the academic Multicore Application Programming Studio, MAPS [5]. At its core, the multicore programming flow follows the structure from the gray area in Figure 2.

Applications are described using the Kahn Process Networks (KPN) programming model [13], using an extension to the C programming language, called C for Process Networks (CPN). A KPN describes computation as processes, which in CPN are simply pieces of C code. These processes communicate via FIFO buffers, called channels. Parallelism can be easily extracted from this so-called

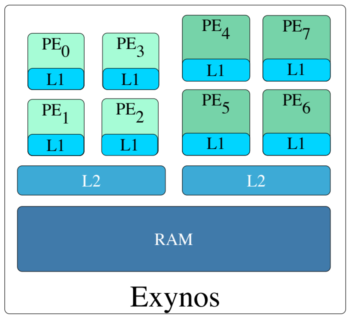


Figure 3: The heterogeneous Exynos architecture.

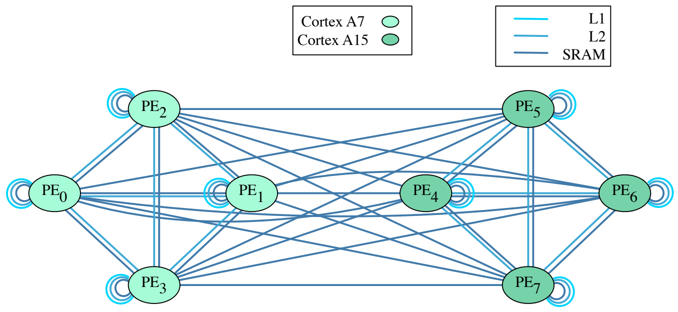


Figure 4: The architecture graph of the Exynos.

dataflow representation of a program. Programs can be either written in this representation directly or obtained from sequential code with automatic parallelizing heuristics.

Hardware architectures, on the other hand, are modeled by meticulously measuring communication times between different parts in the topology to construct analytic models. These models are usually stored in a structured format (xml), where a description of the topology is given alongside the estimated communication times between different parts of the architecture, and the different processing element (PE) types, in the case of an heterogeneous architecture.

These two descriptions are used, combined with either accurate estimations or, when available, measurements from an execution trace to drive heuristics that statically determine a near-optimal static mapping. Here, a mapping refers to an allocation of resources (PEs and memory) for the components of the KPN, i.e., the processes and channels. In this paper, we are not concerned with the mapping heuristics and objectives themselves. The aim of the TETRiS system is to be agnostic to them, leveraging the structure of the target architecture to preserve the properties of the mappings. Thus, the approach could have been implemented in any other task-based multicore mapping system.

2.2 Equivalent mappings

Hardware architectures, even heterogeneous ones, exhibit symmetry to some degree. When all the properties of the PEs and the communication between them are equal, we intuitively expect two mappings to yield the same results. In the following, we will show how this intuitive concept of equality can be described formally and used to automatically find equivalent mappings in MPSoCs.

2.2.1 The Architecture Graph. For the TETRiS approach, we formalize the notion of equivalence of mappings by using graph theory. To recognize the equivalence of mappings, our approach uses a structure called the *architecture graph* to capture the architecture topology [3]. This multigraph $A = (V_A, E_A)$ has a node $v \in V_A$ for every PE in the architecture. A function l labels all the PEs in V_A with their PE type in heterogeneous architectures.

For every communication resource r that can be used between two PEs $v_1, v_2 \in V_A$, the architecture graph has an edge $(v_1, v_2, r) \in E_A$, where r is a label for that communication resource. As an example, consider the Exynos architecture, as illustrated in Figure 3. It has eight ARM PEs, following the big.LITTLE™ principle, with four ARM Cortex A7 PEs (the "little" ones) and four ARM Cortex A15 PEs (the "big" ones). For it, the architecture graph of the Exynos architecture is depicted in Figure 4. The labels for the different communication resources and PE types can be seen in the colors of the edges and nodes in the graph.

This architecture graph can be readily obtained from the architecture description in the SLX Tool Suite, or any similar description which includes the structure of the hardware architecture. In fact, the objective of the architecture graph is to allow us to capture, in a mathematical object, precisely this **structure**: the topology, PE types and the communication resources available. Only by using the formal nature of this architecture graph we can extract the symmetries of the architecture algorithmically, i.e., in an automated fashion.

2.2.2 Extracting Symmetry. In order to extract the symmetry from the architecture graph, we first need to understand how the intuitive concept of symmetry can be described in this graph-based formalism.

Let K be the KPN graph and consider a (valid) mapping $m : K \to A$. We can identify a subgraph $A_m \leq A$ of the architecture graph A which corresponds to m. Namely, the subgraph with precisely those nodes and edges to which a process or channel is mapped, i.e., the image of K under m, $A_m = m(K)$. This means that, for all KPN processes p_1, p_2 connected via the channel c_1, we have $m(p_1), m(p_2) \in V_{A_m}$ and $(m(p_1), m(p_2), m(c_1)) \in E_{A_m}$. Figure 5 illustrates these subgraphs for two example mappings onto the Exynos architecture. On the left of the figure, the mappings are depicted: the tasks A, B, C and D are mapped to different PEs, this is illustrated by drawing the tasks inside the PEs. On the right of the figure, the corresponding subgraphs to each of the two depicted mappings are shown; these are subgraphs of the architecture graph depicted in Figure 4.

By leveraging the corresponding sub-architectures, we can define the equivalence of two mappings. We say two mappings m_1, m_2 are *equivalent* if the function $\varphi : A_{m_1} \to A_{m_2}$, which takes the subgraph of m_1 to that of m_2 is a **graph isomorphism** and, for heterogeneous architectures, **respects the vertex labels** l. Here,

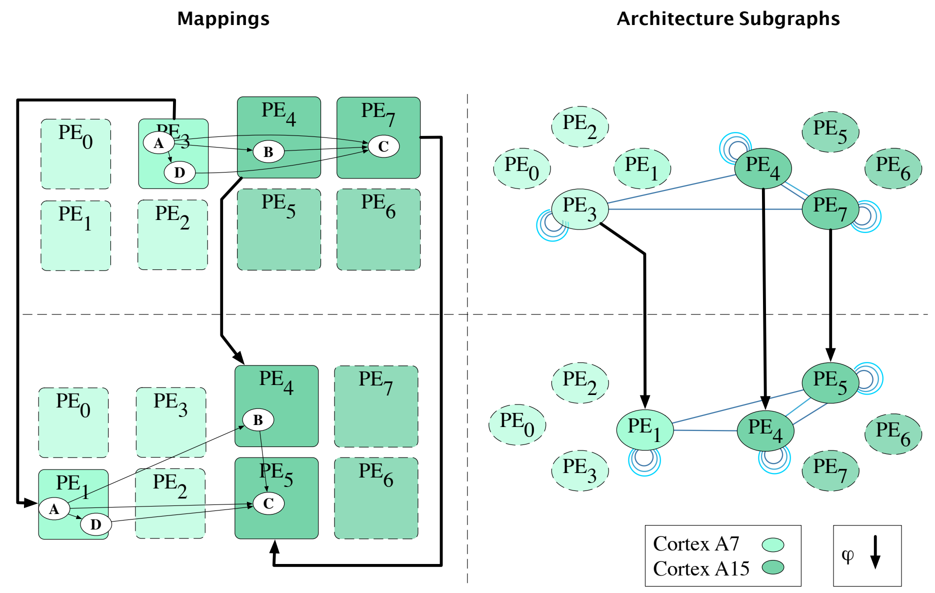


Figure 5: Two equivalent mappings and the corresponding isomorphism of architecture subgraphs.

φ is defined as:

$$\varphi(m_1(p)) = m_2(p) \text{ for all } p \text{ KPN process} \quad (1)$$

$$\varphi(m_1(p_1), m_1(p_2), m_1(c_1)) = (m_2(p_1), m_2(p_2), m_2(c_1))$$
$$\text{for all } c_1 \text{ KPN channel connecting } p_1 \text{ and } p_2 \quad (2)$$

A graph isomorphism is a bijective function φ between the vertex sets of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, such that two nodes are adjacent if and only if their images under φ are adjacent, i.e.:

$$(\varphi(v), \varphi(w)) \in E_2 \Leftrightarrow (v, w) \in E_1$$

An example of this can be seen in Figure 5. In the example, a task graph consisting of tasks A, B, C and D is mapped to the Exynos in two equivalent ways: in the upper diagram, $A, D \mapsto \text{PE}_3, B \mapsto \text{PE}_4, C \mapsto \text{PE}_7$. This is equivalent to the mapping in the lower diagram, namely $A, D \mapsto \text{PE}_1, B \mapsto \text{PE}_4, C \mapsto \text{PE}_5$. This is the case because the isomorphism, $\varphi : \text{PE}_3 \mapsto \text{PE}_1, \text{PE}_4 \mapsto \text{PE}_4, \text{PE}_7 \mapsto \text{PE}_5$, respects the labels of the nodes and edges of the subgraphs, as depicted on the right side of the figure.

If the function φ is a graph isomorphism, then we can think of it basically as a *renaming* of the PEs, which does not alter the structure of the architecture (as captured by the architecture graph A). If, additionally, $V_1 = V_2 = A$, then φ is what is called an *automorphism* of A. The symmetries of hardware architectures can be also studied by analyzing all such functions, using the mathematical theory of *groups* [9]. However, our approach can identify much more symmetry, since it is not restricted to **global** symmetries, as are groups. Similarly, our approach can leverage symmetries that are not intuitive from a geometric perspective. To understand this, consider the example of a regular 4 by 4 mesh Network-on-Chip (NoC), and the sub-architectures depicted in Figure 6. For two sub-architectures to be equivalent, they have to respect the communication pattern in the NoC, i.e., the number of hops (Manhattan distance) between the processors. If we consider a group-theory based solution as outlined above, which only considers global symmetries, we can get **rotations** (or reflections) of the architecture. **Translations**, however, are not such global symmetries: these would be captured in a geometrical approach that understands the geometric shape of the sub-architecture. However, this geometric view is also limited: it cannot consider **transformations** which retain the communication structure but are not geometrically the same.

This concept of equivalence of mappings preserves the performance of the application. By making sure the labels are preserved, the PE and communication types are exactly the same in both mappings. Furthermore, the fact that φ takes all processes mapped onto one PE to another (Eq. 1), and the same for communication (Eq. 2), ensures the execution of the application with both mappings behaves identically.

Several software packages exist, like nauty [17], which accompany a large body of research aimed at finding graph isomorphisms efficiently. For the comparatively small graphs of hardware architecture, this is a negligible computation, especially considering it has to be done once, at design time, since it only depends on the architecture and not on the tasks being mapped.

2.2.3 Changing Mappings at Run-time. We can encode a sub-architecture very efficiently, since a single bit for every involved PE is required. For the Exynos this means a byte for every sub-architecture. The run-time system can then store the PEs which are in use, and using a simple logical and it can find out if there are conflicts.

In order to change a mapping at run-time, however, the isomorphism φ has to be stored. To calculate an equivalent mapping at run-time is thus straightforward: taking the reference mapping m_{ref} that the application wants to use, an equivalent sub-architecture $A_{m_{\text{new}}}$ from the stored sets has to be found, such that $A_{m_{\text{ref}}} \cap A_{m_{\text{new}}} = \emptyset$,

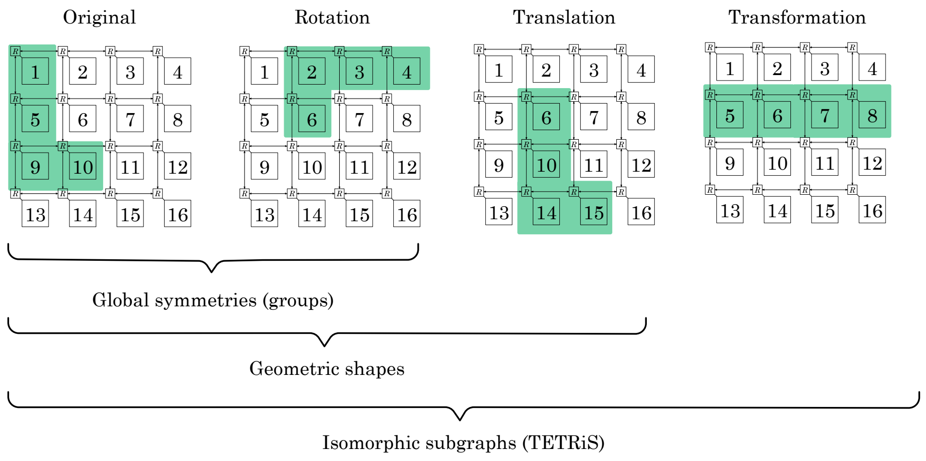


Figure 6: Different intuitive symmetry concepts are captured in the TETRiS approach.

a comparison which can be realized with a single binary bit-wise and. Then, using the stored isomorphism $\varphi : A_{m_{\text{ref}}} \to A_{m_{\text{new}}}$, the new mapping can be obtained by changing every instance of one PE for its image under φ, and similarly for the channels. Altogether, the run-time overhead of selecting a different variant is negligible, since only a search in the pre-calculated database of equivalences is required.

2.2.4 Current Limitations. In general, these simple methods do not scale well with the architecture size. While the storage for subarchitectures scales linearly with architecture size n, the number of isomorphisms grows much faster (in general they are bounded by $n!$). To cope with this, we can define a product between isomorphisms of subarchitectures. Using it, we can compose a small number of these isomorphisms (called a *generating set*) to obtain any possible isomorphism. This uses methods of algorithmic inverse semigroup theory, and their application to software synthesis, which is outside the scope of this paper (see [6, 10]).

The method, as described here, has an additional limitation that is important to note. It will fail if none of the equivalent variants of the mapping can be deployed on the free PEs. In that case, some PEs will be used by multiple applications, which will result in worse and less predictable results. In future work we plan to develop a strategy where non-equivalent variants are also considered, and based on priority, the applications that need the best results are ensured to have an ideal mapping.

2.3 The TETRiS Software Architecture

As implementation, we introduce a software component called the TETRiS architecture, which manages all mapped applications, coordinates their mappings for best utilization of the processors and dynamically changes mappings of running applications when additional programs are started. The TETRiS architecture consists of two components. The first is the global TETRiS server that reads the static mappings. It knows all mapped applications running in the system and assigns cores to tasks according to a user-defined mapping policy. The second component is the TETRiS client library. We do not require modifications to applications that can be scheduled using the TETRiS system. Our client library is loaded by using the `LD_PRELOAD` mechanism present in Linux. At library startup, the aforementioned server is contacted and the program is registered with the TETRiS server. When required, the server may decide to re-map existing applications to accomodate for a new program. At this point the server also selects a mapping for the new application. The client library then interposes calls to functions of the threading library (here *pthreads*) such as `pthread_create` and `sched_set_affinity`. When a call to `pthread_create` is detected, the new thread is reported to the server and mapped to a core according to the application's current mapping policy. The server also records the thread internally for (pontential) future re-mapping.

3 EVALUATION

In this section, we evaluate our approach by using a modern off-the-shelf heterogeneous multicore system and real benchmarks from the signal processing and multimedia domains. With these:

- We check that equivalent mappings are indeed equivalent, by measuring variations in performance and energy consumption between several equivalent mappings.
- We compare the performance and predictability of static mappings, deployed with the TETRiS approach, with the dynamic scheduling performed by the Linux CFS scheduling algorithm in single and multi-application scenarios.
- We analyze the overhead incurred by TETRiS task migrations.

3.1 Experimental Setup

To evaluate our approach, we used the Hardkernel Odroid XU3, a modern off-the-shelf heterogeneous multicore system. The system features an Exynos 5422 big.LITTLE chip with four Cortex-A15 cores and four Cortex-A7. Further, the system features 2 GB of LPDDR3 RAM clocked at 933 MHz. All data used in the experiment was read from and written to a temporary filesystem (unix `tmpfs`). We set fixed frequencies of 1.4 GHz for the A7 (little) cluster and 1.6 GHz for the A15 (big) cluster. The big cluster was not able to sustain higher frequencies, due to the inability of the stock fan

Table 1: Strategies used in the evaluation

Strategy	Uses Tetris	Dynamic Scheduling	Optimization Goal
CFS	x	✓	resource usage
Dyn	✓	✓	N/A
T1	✓	x	CPU time
T2	✓	x	wall-clock time
T3	✓	x	execution time

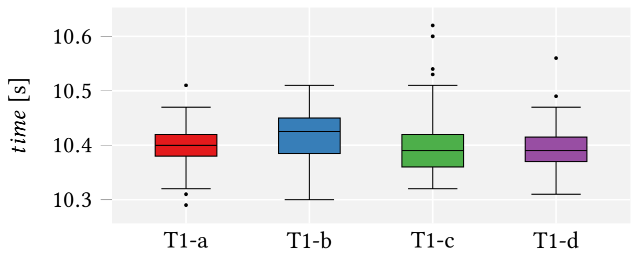
Table 2: Benchmarks and the properties of their dataflow graphs.

Name	Short description	No. of pr	No. of ch.
AF	Stereo frequency filter.	8	8
MIMO	Multiple-input multiple-output orthogonal frequency-division multiplexer	10	16

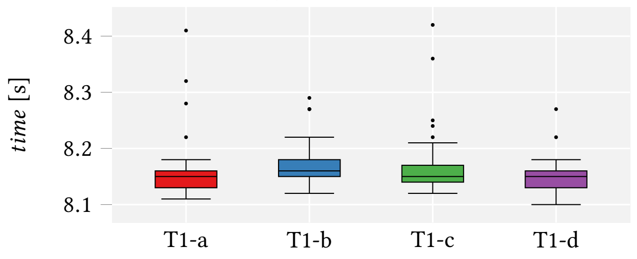
to adequately cool the chip. We used two different applications for benchmarks. Table 2 gives a summary of the properties of the corresponding dataflow graphs, as well as a brief description of both applications. As input to the AF benchmark, we provided a 36 s, 16 bit file at 48 kHz as an input. For the MIMO benchmark, we used randomly generated packets, 15000 in our setup.

For our benchmarks we measured two time metrics, wall-clock time and CPU-time, as well as the energy consumed by the system. Wall-clock time is the time as (humanly) perceived from start to finish, while CPU-time is the total time all PEs used during execution. Time was measured using the Unix `time` utility, while energy was measured by accessing the on-board energy sensors featured on the Odroid XU3. These INA-231 sensors are connected via the I2C bus and measure the energy at the voltage regulators of individual components. They provide detailed energy data for the big cluster, little cluster, memory and GPU of the system, sampled at 10 Hz. In our experiments, we report the aggregate of those four values, due to space constraints.

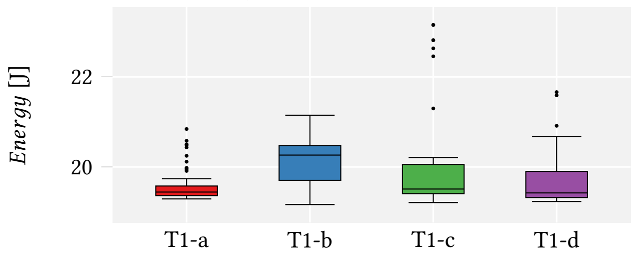
In all scenarios, we used up to four instances of the applications. To fit four instances with the TETRiS approach, we generated static mappings for the two benchmarks using two PEs, one "big" and one "little" (ARM_0 and ARM_4, in Figure 3). Since in this particular example the numbers are tractable, we exhaustively tested all possible mappings on these two PEs to find the mapping which yielded the best CPU time (T1), and the best wall-clock time (T2). We used the compiler flow described in Section 2.1 to generate a third static mapping (T3). For each fixed mapping, T1, T2, and T3, we generated symmetric mappings (e.g., T1-b, T1-c, etc.) that use different big and little cores. We did so using the approach outlined in Section 2.2.2. We compared our static mappings against the Linux scheduler (CFS), as well as an additional strategy: we let the Linux scheduler only use the same two cores as in the static mappings (also transformed with TETRiS for the multi-application scenarios) but schedule the processes dynamically with these constraints (Dyn). This allows the Linux scheduler to dynamically move tasks between the "little" and "big" core, providing better utilization but



(a) CPU time



(b) Wall-clock time



(c) Energy

Figure 7: Four equivalent mappings for mapping strategy T1 for the AF benchmark.

also leading to less predictable execution times and energy usage. The different strategies are summarized in Table 1. All benchmarks were run 50 times for each configuration reported and we present the results as boxplots.

We evaluated the TETRiS approach from three different perspectives. First, we show the performance variation between symmetric mappings in Section 3.2. We evaluate performance and predictability in single and multi-application scenarios in Section 3.3. Finally we demonstrate the overhead of a switching event and its implication on power usage in Section 3.4.

3.2 Symmetry

In the first experiment, we check that mappings from the same equivalence class have no significant deviation in terms of the performance and energy usage. In other words, we check that equivalent mappings are, indeed, equivalent. To evaluate that, we ran the AF benchmark with each of the mappings T1-a, T1-b, T1-c, T1-d. These are all different variations of the mapping T1 according to our TETRIS principle, each with a different pair of PEs. Figure 7 shows the CPU time and energy consumption of each mapping strategy. The relative standard deviations of the average values between the symmetric mappings are the following: 0.13 % for CPU time, 0.12 % for wall clock time, and 1.33 % for energy usage. These deviations are negligible, which allows us to rely on the performance equivalence of the symmetric mappings. In the following sections, we do not specify which particular mapping we use among equivalent ones.

3.3 Predictability

To show predictability, we tested three different scenarios. The first one is a single instance of Audio Filter (1×AF) running alone; the results are shown in Figure 8. The second scenario consists of four instances of Audio Filter (4×AF), with results shown in Figure 9, whereas the third scenario combines the two benchmarks by running two instances of AF and two instances of MIMO (2×AF 2×MIMO). The results of this third scenario are depicted in Figure 10. All scenarios were evaluated for the mappings T1, T2, and T3, and compared with the Dyn strategy and the CFS scheduler.

The results illustrate that using fixed mappings guarantees stable and predictable execution times in terms of both, resources used (CPU Time) and actual time passed (wall clock time). The energy usage, albeit higher than when using dynamic scheduling, is also significantly more stable and predictable. The higher energy usage in this case is understandable, since for this benchmark, CFS uses four times as many physical resources as the TETRiS-based strategies. The MIMO benchmark has a higher amount of inter-process communication. The consequences of this can be seen in the time fluctuations in Figure 10.

Since the Exynos architecture is based on a single bus and a shared-memory model, contention is unavoidable. We see that, while TETRiS allows an isolated view of the applications resulting in predictable execution times for a given system-load, the effects of contention from communication still affect the overall execution time. To better compare this, and to quantify the differences in predictability, Table 3 gives the means and variances for the wall-clock time and energy consumption of all three scenarios. For space reasons, only the results for the static mapping strategy T3 are presented, which is the one obtained from the tool. From the table we can see that, by using TETRiS, the variance of the energy is reduced by a factor of up to 83 compared to CFS. For the wall-clock time, this factor reaches up to 510.

3.4 Remapping

As changing a mapping during runtime might incur overhead due to the cache state and additional system calls performed, we wanted to quantify this effect. To that end, we measured the overhead for each of the mapping strategies when switching from one symmetric mapping for AF to another. The results are illustrated in Figure 11. In it, a switch to a symmetric mapping is performed after the application was running for roughly half its expected execution time and the wall-clock time, CPU time and energy consumption were measured. We normalize the results to the average wall-lock time, CPU time and energy for the same application while running without remapping (compare to Figure 8) and show a boxplot of the values from 50 runs.

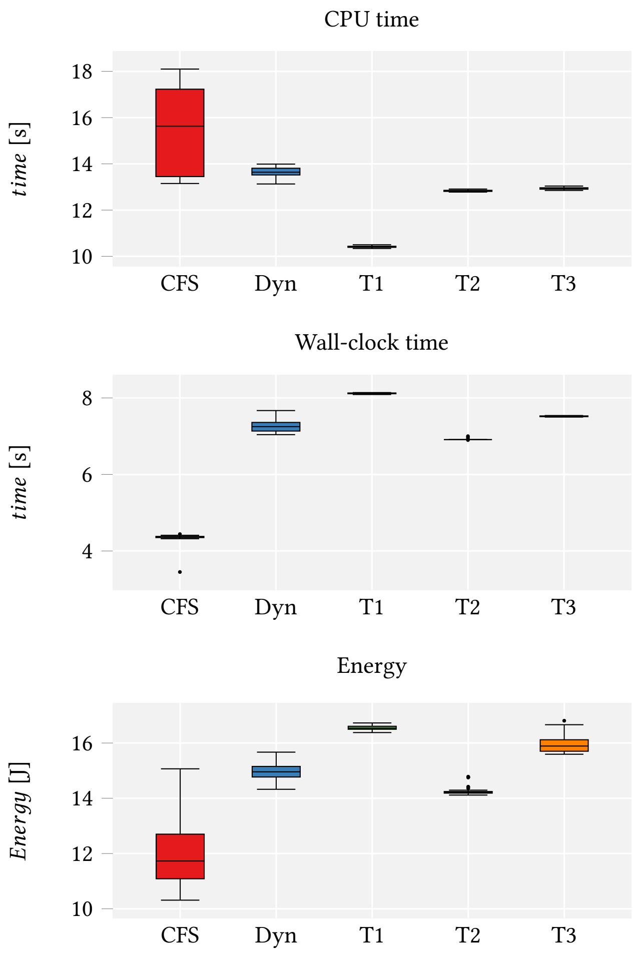


Figure 8: Audio Filter (AF) running alone in the system scheduled on two cores (CFS uses all PEs).

Table 3: Means and Variances of the Different Experiments (all values for Audio Filter)

Scenario	Strat.	Exec. time [s]		Energy [J]	
		Mean	Var	Mean	Var
1×AF	CFS	4.35	$1.7 \cdot 10^{-2}$	12.07	$1.2 \cdot 10^{0}$
1×AF	Dyn	7.25	$2.1 \cdot 10^{-2}$	14.95	$9.6 \cdot 10^{-2}$
1×AF	T3	7.52	$9.4 \cdot 10^{-5}$	15.96	$8.9 \cdot 10^{-2}$
4×AF	CFS	7.96	$1.3 \cdot 10^{-1}$	43.79	$2.9 \cdot 10^{0}$
4×AF	Dyn	8.35	$9.3 \cdot 10^{-2}$	44.81	$3.9 \cdot 10^{0}$
4×AF	T3	8.17	$2.7 \cdot 10^{-3}$	44.35	$6.9 \cdot 10^{-1}$
2×AF, 2×MIMO	CFS	6.31	$4.3 \cdot 10^{-1}$	37.00	$1.6 \cdot 10^{1}$
2×AF, 2×MIMO	Dyn	7.59	$6.3 \cdot 10^{-2}$	41.08	$2.6 \cdot 10^{1}$
2×AF, 2×MIMO	T3	7.65	$8.5 \cdot 10^{-4}$	34.05	$1.9 \cdot 10^{-1}$

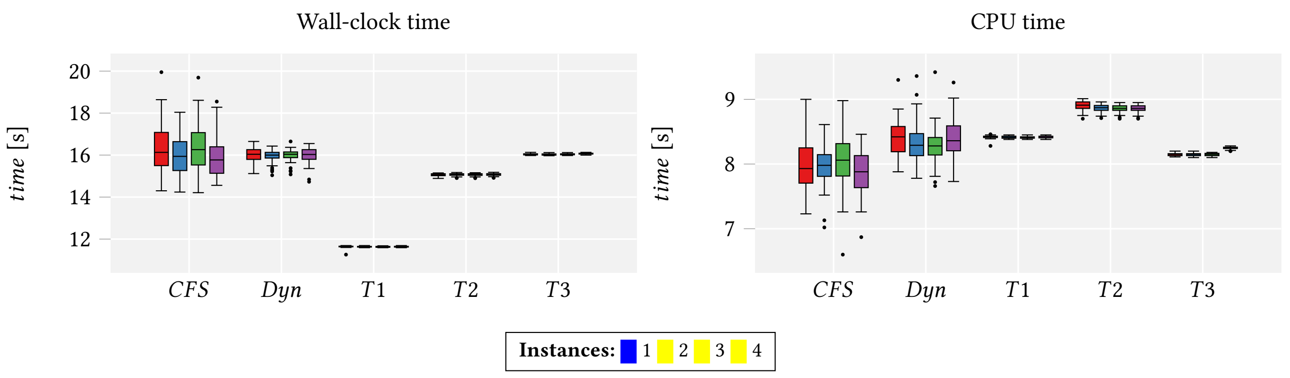


Figure 9: Four instances of Audio Filter (AF) running at the same time.

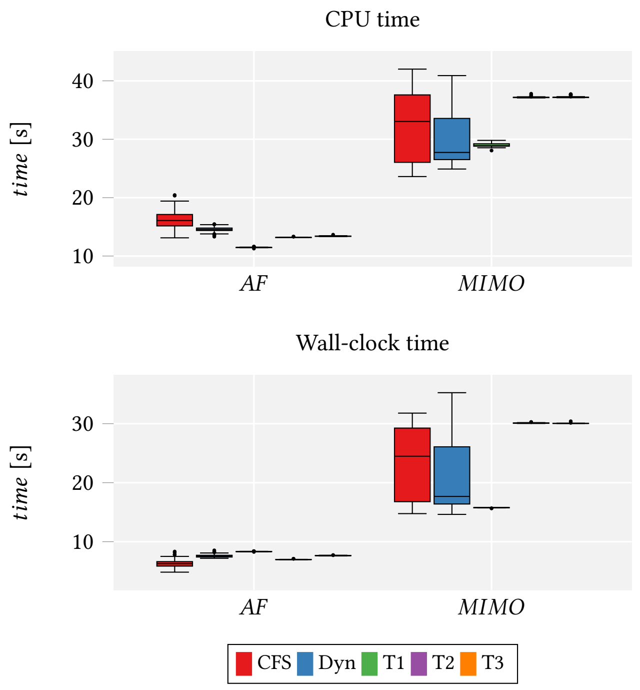


Figure 10: Two instances of Audio Filter (AF) running together with two MIMO instances.

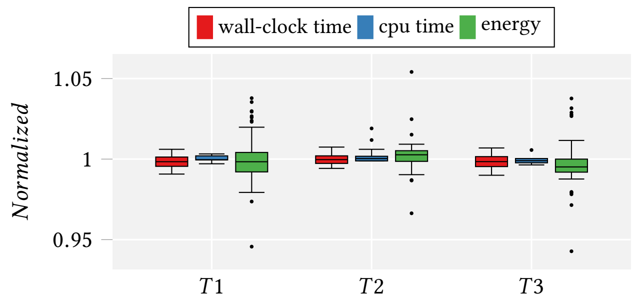


Figure 11: Switching overhead

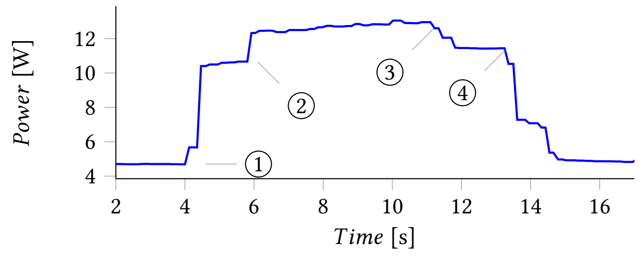


Figure 12: External power trace for a switching event.

In an ideal case, where we do not witness any overhead, we would expect a box plot with minimal variance, centered at the value 1. The figure shows that this is roughly the case for all of our mapping strategies. The average overhead is between 0.017 % in terms of CPU time when using strategy T2 and 0.29 % in terms of energy when using mapping strategy T3. The distribution of the values (quartiles, outliers) is comparable to the distribution of the measurements obtained when running the application without switching. Overall, the overhead of switching static mappings running TETRiS at runtime is negligible for our benchmarks.

To better understand remapping, Figure 12 illustrates the power trace of a typical switching scenario. It shows a four-core mapping of AF running ① and, after 2 seconds, three more instances being added ②. TETRiS then remaps all AF instances to a two-core mapping utilizing one big and one little core for each program to accommodate all of them equally. After a short while, the orignal AF instance finishes ③ and then the remaining three AF instances terminate ④. The trace shown was collected with an external power meter to avoid influencing the running applications by the measurement. Accordingly, the power consumption contains more components and is higher than the value reported by internal power sensors used in the previous figures.

We also measured the overall overhead of using the TETRiS server. The average overhead for wall-clock time was 0.25 %. The maximum overhead we observed was 1.8 %.

4 RELATED WORK

The problem of predictable scheduling in MPSoCs has attracted much interest from the research community, and resulted in a large body of research work. In particular, several methods have been developed which use static (compile-time or design-time) and dynamic (run-time) approaches, or a combination of both. Various methods, like those presented in [4, 14, 28], address multi-application scenarios in a static fashion, by knowing the different applications at compile time. These methods do not scale, since the amount of possible scenarios grows exponentially with the number of applications. To deal with these scalability issues, research has drifted towards so-called hybrid approaches, where the TETRiS methodology can be classified. An extensive survey of mapping strategies can be found in [24]. In it, most of the hybrid strategies presented are restricted to fixed/homogeneous platforms or also suffer from the same issues mentioned above, that all the scenarios need to be known at compile time.

Research methods have resorted to abstractions and modularity to support different applications without knowing them all at compile-time. For example, the authors in [18] propose a two-tiered approach, where a mapping is first calculated to virtual processors and subsequently at run-time to physical processing elements (PEs). The presented algorithm for mapping to physical PEs, which has a goal similar to the TETRiS approach is a special-tailored heuristic which tries to minimize the number of links used. It assumes a heterogeneous system in terms of PEs and a particular network topology. Additionally, the goal of the heuristic is to minimize a particular objective, not to preserve the properties of the original mapping (to virtual processors). A similar two-tiered approach has been proposed by the authors in [22]. This approach uses a divide-and-conquer strategy to leverage partial solutions in order to calculate a good mapping at run-time. However, this hybrid approach calculates the mapping at run-time with a greedy heuristic, and is bound to yield less predictable results. It also optimizes for particular goals, like load-balancing and minimal migration, which limits the usefulness of the approach to scenarios where this is desired. Similarly, the approach presented in [26] uses a costly backtrack algorithm and targets specific goals, by minimizing the communication distance in the mappings.

Finally, the approach put forward in [27], which aims to isolate cores motivated by security, shares some of the core ideas with the TETRiS approach. In it, the authors define the concept of shapes, which in a sense encapsulates a subset of the mapping equivalences defined in this paper. This method is limited to very regular architectures (rectangular 2D-mesh NoCs), and will fail on most other architectures, bus-based ones, or ones with complex, hierarchical and heterogeneous structures, where the (euclidean) geometric intuition of shapes breaks down. Moreover, this approach cannot detect equivalent mappings that are not an affine transformation of the original (cf. Figure 6), nor will it work for non-contiguous regions in the NoC. The paper does not also consider dynamic remapping of applications. Apart from these limits, however, finding a mapping with this hybrid approach is comparable to the TETRiS mapping transformation.

To the best of our knowledge, the TETRiS approach is the first hybrid solution that is completely agnostic to the static mapping decisions, works for arbitrary architectures, including off-the-shelf hardware, and supports Linux, all of which makes it versatile and modular. Additionally, the TETRiS approach considers equivalences in mappings explicitly, by transforming a fixed mapping instead of calculating one from partial solutions. It has the advantage of having basically no run-time overhead.

5 CONCLUSIONS

In this paper we have introduced a novel method for dealing with static mappings in multi-application scenarios. It allows remapping and migrating in a way that preserves the benefits of a static mapping, like cache coherence or less resource utilization, resulting in more predictable outcomes. While not a formal guarantee of performance, results of running our TETRiS server using two benchmarks on a modern heterogeneous multicore showed significantly more predictable results, especially in multi-application scenarios. While TETRiS is agnostic to mapping strategies, our experiments showed how, when communication contention is involved, the optimal mappings for the single-application scenario are not necessarily the optimal mappings in multi-application scenarios. Nevertheless, the TETRiS approach outperformed the Linux scheduler CFS in terms of predictability of the execution time by a factor of 510 and a factor of 83 for the energy efficiency in our benchmarks.

ACKNOWLEDGMENTS

This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Center HAEC and the the Center for Advancing Electronics Dresden (cfaed). We thank Silexica for making their SLX Tool Suite available to us.

REFERENCES

[1] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82, 2014.

[2] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. Van Meerbergen. Predictable embedded multiprocessor system design. In *International Workshop on Software and Compilers for Embedded Systems*, pages 77–91. Springer, 2004.

[3] J. Castrillon, A. Tretter, R. Leupers, and G. Ascheid. Communication-Aware Mapping of KPN Applications onto Heterogeneous MPSoCs. In *DAC '12: Proceedings of the 49th annual conference on Design automation*, 2012.

[4] J. Castrillon, R. Velasquez, A. Stulova, W. Sheng, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. Trace-based kpn composability analysis for mapping simultaneous applications to mpsoc platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 753–758. European Design and Automation Association, 2010.

[5] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. Maps: an integrated framework for mpsoc application parallelization. In *Proceedings of the 45th annual Design Automation Conference*, pages 754–759. ACM, 2008.

[6] J. East, A. Egri-Nagy, J. Mitchell, and Y. Péresse. Computing finite semigroups. *arXiv eprints*.

[7] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual Design Automation Conference*, pages 264–265. ACM, 2007.

[8] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[9] A. Goens and J. Castrillon. Analysis of process traces for mapping dynamic kpn applications to mpsocs. In *Proceedings of the IFIP International Embedded Systems Symposium (IESS)*, Foz do Iguaçu, Brazil, Nov. 2015.

[10] A. Goens, S. Siccha, and J. Castrillon. Symmetry in software synthesis. *arXiv e-prints*, Apr. 2017.

[11] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1–8, 2011.

[12] A. Hansson, K. Goossens, M. Bekooij, and J. 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.

[13] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[14] S.-h. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele. Static mapping of mixed-critical applications for fault-tolerant mpsocs. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[15] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and Y. Ha. Analyzing composability of applications on mpsoc platforms. *Journal of Systems Architecture*, 54(3):369–383, 2008.

[16] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[17] B. D. McKay and A. Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

[18] O. Moreira, J. J.-D. Mol, and M. Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1557–1564. ACM, 2007.

[19] 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 *Proceedings of the 45th annual Design Automation Conference*, pages 574–579. ACM, 2008.

[20] A. Olofsson. Epiphany-v: A 1024 processor 64-bit risc system-on-chip. *arXiv preprint arXiv:1610.01832*, 2016.

[21] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[22] W. Quan and A. D. Pimentel. A hierarchical run-time adaptive resource allocation framework for large-scale mpsoc systems. *Design Automation for Embedded Systems*, pages 1–29, 2016.

[23] Silexica. SLXMapper, 2016.

[24] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013.

[25] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping applications to tiled multiprocessor embedded systems. In *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on*, pages 29–40. IEEE, 2007.

[26] A. Weichslgartner, D. Gangadharan, S. Wildermann, M. Glaß, and J. Teich. Daarm: Design-time application analysis and run-time mapping for predictable execution in many-core systems. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[27] A. Weichslgartner, S. Wildermann, J. Götzfried, F. Freiling, M. Glaß, and J. Teich. Design-time/run-time mapping of security-critical applications in heterogeneous mpsocs. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, pages 153–162. ACM, 2016.

[28] D. Zhu, L. Chen, S. Yue, T. Pinkston, and M. Pedram. Providing balanced mapping for multiple applications in many-core chip multiprocessors.