On the Representation of Mappings to Multicores

Andres Goens, Christian Menard, Jeronimo Castrillon
Center for Advancing Electronics Dresden (cfaed),
Chair for Compiler Construction, TU Dresden
Dresden, Germany
Email: {first.last}@tu-dresden.de

***Abstract*—Application requirements for embedded systems are growing rapidly, as is the complexity of systems designed to execute them. A common abstraction used to tame this growing complexity is that of a mapping, which assigns parts of an application to different hardware resources. Modern flows need to explore an intractably large design space of mappings, and be able to quickly find near-optimal mappings for different objectives, sometimes at runtime. With systems featuring thousands of cores in the near horizon, we need methods to make this exploration step truly scalable. In this paper we argue that the mathematical representation of a mapping is central to achieve this. We present different representations and how these could be applied to different contexts and objectives, like complex design-space exploration meta-heuristics or efficient runtime systems.**

I. Introduction

Traditionally, embedded systems have had a very specific development cycle. A set of requirements is transformed into an integrated circuit with hardware and software tailored to that specific application, to fulfill these precise requirements. However, with shorter time-to-market cycles, increasing requirements and the ubiquity of cheap and powerful devices, this landscape is changing rapidly. Modern embedded systems are not static anymore. The behavior and application requirements change at a rapid pace, forcing systems to be able to adapt to new and more complex workloads that were not envisioned at design-time. Like its software counterpart, embedded hardware is increasing in complexity, too. Heterogeneous multi processor systems on chip (MPSoCs), with technologies like Network on Chip (NoC) and distributed memory are becoming standard. It is thus a very significant problem to program these modern and complex systems.

A body of approaches which aims at tackling this problem uses a selection of ideas collectively called *software synthesis*[2], [5], which aim to lower an abstract representation of an application to execute on a specific target. In analogy to high-level synthesis, at the center of the software synthesis approach is a design-space exploration step that looks for an optimized implementation of a high-level application specification into platform-specific primitives. This includes a central abstraction, called *mapping*, where logical tasks are assigned to system resources in a way that optimizes for performance and/or energy consumption. An important difference is, however, that in software synthesis, the mapping decision can be partially or completely deferred to runtime, making speed in the exploration even more critical. With the growing complexity of both system architectures and applications, it becomes increasingly crucial that this design-space exploration is scalable. The number of mappings for a modest 27-task application to an architecture with 1024 cores, like the one described in [20] is over 10^{81}, more than the estimated number of atoms in the observable universe. Architectures with thousands of cores are already being designed and produced today [20], [6], and the trend suggests these numbers will only continue to grow.

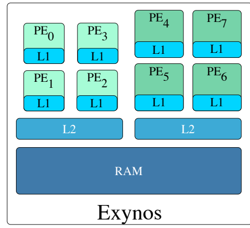
We argue that simply using domain-knowledge to reduce the design-space, while extremely useful, is not sufficient. In order to tame the dauntingly large design-spaces, smart meta-heuristics must leverage the inherent structure of the problem to navigate them. Of central importance for any meta-heuristic exploring the design-space of mappings is its representation, which traditional approaches only address marginally.

In this paper we address the representation of mappings in software synthesis explicitly. We show common design-space reduction techniques, like using the problem's symmetries, and explain how the representation of mappings can be used to reflect this. We also argue for the need of endowing mapping spaces with a metric for a large variety of exploratory heuristics which rely on topological features of the problem space, and show how to combine these approaches. Finally, we present numerous ways the different representation of mappings can be applied to state-of-the-art software synthesis techniques. These applications include find mappings quickly at runtime, explore large design spaces intelligently at compile-time or even find specially useful properties of mappings which emerge from a particular representation.

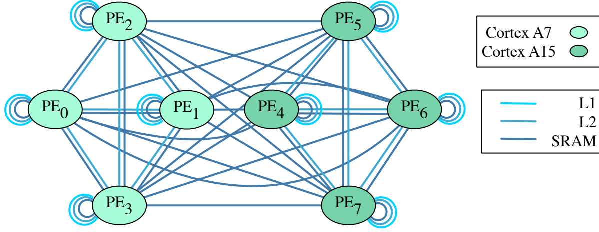
II. Background: Mappings to Multicore Architectures

The process of *software synthesis* is an umbrella term for a series of approaches which leverage structural information of an application and its target architecture to generate code in a way that focuses on a given objective metric (or multiple objectives). Common of these approaches is the usage of an abstraction called *mappings*. Intuitively, a mapping is an assignment of hardware resources to the different parts of the application, both in terms of computation as well as data communication.

We will now formally define mappings as a mathematical structure. For this, we endow applications with a graph-like structure, as is commonly done in literature. Formally, an



(a) The Exynos Architecture



(b) The Exynos Architecture Graph

Figure 1. An Architecture and its Corresponding Architecture Graph.

application graph $K = (R, C)$ is a graph representing the architecture, where vertices R represent computational tasks, and the edges C represent either explicit data communication (channels) or logical dependencies between the computational vertices. This can be, e.g. a task graph (like the ones in [3]), or a dataflow graph in a particular dataflow model of computation, e.g. Synchronous Data Flow [15] or a process network following the Kahn-MacQueen execution semantics [14].

Similarly, for a given hardware architecture, we define a *topology graph* $T = (P, E)$ as follows: For every processing element (PE) in the architecture, we have a node $p \in P$. For every communication resource (e.g. shared cache, NoC router, etc.) that allows **direct** communication between two PEs p, q we add a node (p, q). If the memory subsystem is heterogeneous, in the sense that different direct communication methods will have different communication times, then we add weights to the edges (p, q) that correspond to these heterogeneous communication times. This topology graph is useful for reasoning at an intuitive level. However, for formalization, we derive another graph from it, which we call the architecture graph. For a topology graph $T = (P, E)$, the corresponding *architecture graph* A is a multigraph with vertex set P. A path $c = p_1, \ldots, p_k$ in the topology graph represents a way of communicating between the PEs p_1 and p_k. We do not want to model communication going in circles, and thus assume that the communication path c visits every vertex at most once. Thus, for every such communication path c, which visits every vertex at most once, we have an edge in the architecture graph A describing the costs of communicating via this path. Similarly, the architecture graph is usually trimmed to include only communication paths that are not strictly worse (not only considering latency, but also memory utilization, etc.) than another paths between two points. An example of a heterogeneous system, exynos [12] and its corresponding architecture graph A are depicted in Figure 1.

Having formally defined applications and architectures, we can define a mapping. A *mapping* m is a morphism of graphs $K \to A$ from the graph of the application to the architecture graph of the target architecture. This means that the communication channel or dependencies from the application are also mapped to communication resources in the architecture. Similarly, let $f : \{\text{Mappings}\} \to \mathbb{R}^n$ be an objective function, that gives some desirable objective(s) for a given mapping. In this context, the components of $f(\cdot)$ could represent, for example, the execution time for a particular input stimulus, the worst-case execution time, or the average energy consumption. The mapping problem can thus be expressed in simple mathematical terms: it is to find Pareto-optimal points $f(m), m \in M$ over the set M of all valid mappings (e.g. excluding those which allocate more data to a memory than is available, or violate system constraints like the number of applications that can be scheduled in a PE). The most common case is for $n = 1$, where this reduces to maximizing or minimizing an objective, like performance or energy consumption. This problem has been studied extensively in different forms and with different objectives, a survey can be found in [24].

III. Representations

As discussed in Section II, the mapping problem is subject of a large body of research, aiming for different objectives, like performance, throughput, energy efficiency or resource utilization. Most strategies and heuristics explore the exponentially-growing design space of mappings. However, most give little attention to the way mappings are represented for this exploration.

Problems that are NP-hard, with intractably large spaces of possible solutions, can still be solvable (or approximable) in practice for the instances that are interesting to the applications. Both in academia and industry, problems like SAT, ILP or the traveling salesman are solved or approximated routinely to address very real problems, like hardware verification or planning airplane routes. Central to solving these problems in practice are heuristics that leverage very concrete structure in the design space, like the conflict-driven clause learning (CDCL) methods in SAT [23], which builds upon search trees and implication graphs. In a very direct way, these heuristics are only possible because of the representations used that allow to navigate the space in an intelligent way. In particular, stating the problem in terms of these abstract representation allows to build on top of them. It is debatable if heuristics like CDCL would have been possible without thinking of the problem in terms of these representation.

We argue that we can learn from these neighboring domains for the mapping problem. If mapping spaces are to be explored efficiently for practical instances in future architectures, neither design-space reductions nor simple mapping heuristics that just guess a mapping that should be good, without exploring the space, are going to be sufficient. Furthermore, generic meta-heuristics will not scale with the exponentially-growing design space, if not very carefully refined for the problem. In any case, for a sophisticated algorithm to explore the design

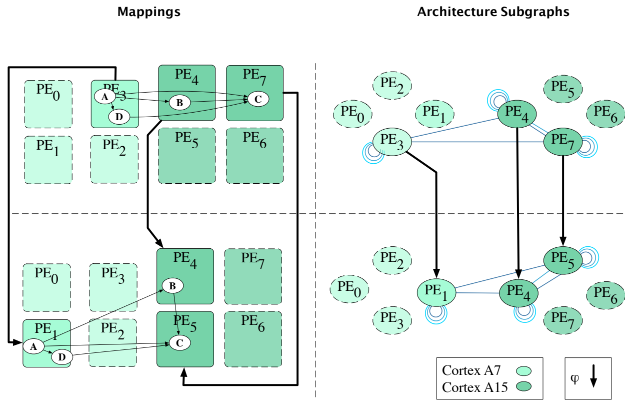


Figure 2. An Example of Two Mappings Equivalent by Symmetries.

space of mappings, be it at compile-time or at runtime, a particular focus on the representations of the mappings is imperative. The difficulty arising from this insight is not if this is true, but rather, what representations might indeed be most useful. In this section, we propose some candidates for the representation of mappings we believe might be useful and explain the reasoning behind them.

A. Simple Vector Representation

The simplest representation of mappings, which is used by most heuristics and meta-heuristic-based approaches, is the naive vector representation of a mapping. Consider an application $K = (R, C)$ mapped to an architecture with architecture graph A, via the mapping m. Then, for every task/process $r \in R$, $m(r) \in P$ defines a PE in A that will execute r. Similarly, a communication channel $c = (r, r') \in C$ is mapped to a hardware communication path $m(c)$ in A. The *simple vector representation* of m is thus the $|R| + |C|$-tuple $(m(r_1), \ldots, m(r_{|R|}), m(c_1), \ldots, m(c_{|C|}))$, where $R = \{r_1, \ldots, r_{|R|}\}$ and $C = \{c_1, \ldots, c_{|C|}\}$. Sometimes, communication will be ignored and the representation vector for m is taken to be $(m(r_1), \ldots, m(r_{|R|}))$.

B. Symmetries

The mapping problem exhibits much symmetry. This symmetry can be leveraged explicitly by algorithms, using e.g. the simple vector representation and methods from group theory [9], [11]. However, by using a symmetry-aware representation directly, algorithms can explore a fundamentally smaller design-space, often improving the quality of the results [21]. In this section we will first introduce the concepts of symmetries in mappings and show how a representation can leverage them directly.

Mathematically, symmetries are usually described as *transformations* that can be applied to an object, leaving it unchanged. A common approach of this is to use group theory for describing these transformations. In the mapping problem, symmetries of the architecture (and application) induce also symmetries on a mapping [11]. Consider the two mappings depicted on the left-hand side of Figure 2. Intuitively, we would expect them to have the same execution behavior if nothing else is running on the platform. The processes use the same types of processors, and communication patterns between processes are also identical. This can be formalized as an isomorphism φ of (sub-)graphs [10], as shown on the right-hand side of Figure 2. All such isomorphisms of subgraphs have the structure of an inverse semigroup and can be leveraged algorithmically [11].

These symmetries define an equivalence class on the set of mappings. Two mappings are equivalent if there is a symmetry that takes one mapping to the other, like the two mappings on Figure 2. Using the argumentation above, it is easy to see that the behavior of two equivalent mappings should be identical for any a priori consideration. Indeed, any system-level simulator will report the same results for equivalent mappings. This equivalence of mappings can be leveraged both at compile-time [11] and at design-time [10], as will be explained in Section IV.

There are multiple ways to have a representation that uses symmetries. A simple way is using the naive vector representation and using a *canonical representative* for every equivalence class. A canonical representative is a unique element of the equivalence class, such that two classes are identical if and only if their canonical representatives are. Such representatives can be computed by using e.g. a lexicographical ordering on the naive vector representation. Algorithm 1 describes an algorithm to efficiently find a canonical representative of a mapping (via lexicographical ordering). In it, the set of group elements $S = \{g_1, \ldots, g_s\}$ is called a generating set if any element $g \in G$ can be written as a product of elements of S, i.e. $g = s_1 \cdot \ldots \cdot s_n$, where $s_i \in S$ for all $i = 1 \ldots n$. We say that S *generates* G and write $\langle S \rangle = G$. We define the generating set S to be *strictly order-preserving*, if for any two mappings m, $m' = gm$, if $m' < m$, then there exists a word $s_1 \cdot \ldots \cdot s_n$ in S such that $g = s_1 \cdot \ldots \cdot s_n$ and $s_i(s_{i+1} \cdot \ldots s_n)m < (s_{i+1} \cdot \ldots s_n)m$ for all $i = 1 \cdot n$. For example, the group of all permutations on n points, S_n, can be generated by the transpositions $T := \{(ij) \mid i \neq j \in \{1, \ldots, n\}$ which swap exactly two points. This set is strictly order preserving for the action on mappings (see [11] for more details on this).

Algorithm 1 Finding Canonical Representatives.

input: A mapping m, a strictly order-preserving generating set S, with $\langle S \rangle = G$.
output: A mapping $m_{\text{canonical}} = gm$ with $m_{\text{canonical}} < m'$ for all $m' \in Gm$

```
1: F ← {m}
2: F_old ← ∅
3: while F ≠ F_old do
4:     F_old ← F
5:     for all s ∈ S do
6:         for all m' ∈ F do
7:             if sm < m then
8:                 F ← sm
9:     F ← {min_{m'∈F} m'}
   return min_{m'∈F} m'
```

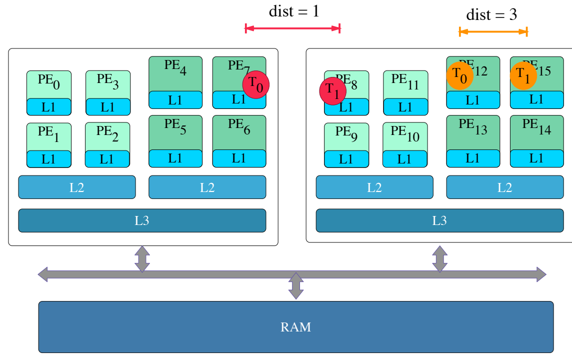


Figure 3. The Problem with Mapping Distance in the Simple Vector Representation.

The canonical representatives define a subset of the set of mappings, but are, in a sense, still the naive vector representation. A better way to leverage symmetries in the representation uses embeddings and will be discussed in Section III-C.

C. Metric Spaces and Embeddings

A different problem with the simple vector representation arises when considering algorithms that utilize concepts of distance or locality, like many approaches in design-space exploration, e.g. evolutionary algorithms or methods based on geometric principles [26], [13]. To illustrate the problem, consider the architecture depicted on Figure 3. It consists of two Exynos chips connected via an off-chip interconnect (bus). Each of the two chips has two clusters of PEs, in which PEs share the L2 cache, and both clusters share an L3 cache.

Intuitively, the distance between two PEs corresponds to the communication latency for accessing data used by both PEs. Thus, the distance between two tasks mapped on the same PE is minimal, on the same cluster is larger but smaller than in a different cluster on the same chip, which is still smaller than the distance to a PE in a different chip. Using the naive vector representation, this distance is very distorted. Indeed, between PE_7 and PE_8, the distance of 1 is just a third of the distance between PE_{12} and PE_{15}. However, the first pair is in different chips, while the second pair shares an L2 cache. Thus, an approach using the the naive vector representation would consider this distorted distance.

To solve this problem, we propose to endow the space of mappings with a metric-space structure. A *metric space* $(\mathcal{M}, d)$ is a structure corresponding of a set $\mathcal{M}$ and a distance function d, which captures the intuitive notion of distance between objects. The set of PEs P can be readily made into a metric space by defining $d(p,p) = 0$ for all $p \in P$, and $d(p,q)$ to be proportional to the (expected) latency between the PEs p and q. In our example, we set the distance of two PEs sharing an L2 cache to be 5, for those sharing an L3 cache, we set 20 and for those only able to communicate via off-chip memory, we set d to be 100. In some cases, e.g. when burst transfers are possible, or when contention in the interconnect is high, this static cost model might be too simplistic. In future work we plan to investigate how to best represent these scenarios.

A very useful tool when describing (finite) metric spaces is the *distance matrix*. For a metric space $(\mathcal{M}, d)$, its distance matrix G is defined as $(G)_{m_i,m_j} = d(m_i, m_j), m_i, m_j \in \mathcal{M}$, and is an element of $\mathbb{R}_{\geq 0}^{|\mathcal{M}|\times|\mathcal{M}|}$.

Having defined a structure as a metric space $\mathcal{M} = P$ on the hardware architecture, it is simple to extend it to mappings, i.e. $\mathcal{M} = M$. Just as in the naive vector representation, we consider M as tuples in $\mathcal{M} = M = P^{|R|}$, and extend this to be a metric space by defining the distance function in a fashion similar to the real L_p norms:

$$d_p(\ (q_1, \dots, q_{|R|}), (\tilde{q}_1, \dots, \tilde{q}_{|R|})\ ) := \sqrt[p]{\sum_{i=1}^{|R|} d(q_i, \tilde{q}_i)^p} \quad (1)$$

The case for $p = 1$ is usually called Manhattan distance, or for $p = 2$ it is a generalization of the euclidean distance. In our example we chose $p = 1$, since just as in the streets of Manhattan, communication packages have to travel through the different components in the communication network and thus the cost are additive.

In order to actually leverage this representation as a metric space, we need an *embedding* into an euclidean vector space, i.e., a representation of the points in the metric space as real vectors, endowed with the euclidean distance. An *isometry*, meaning the distances are the same in both representations, would be ideal. However, this does not exist in general for a finite metric space (see [18], Chapter 15). Instead, a *low-distortion* embedding can be found. A distortion D of such an embedding $\iota : (\mathcal{M}, d) \hookrightarrow (\mathbb{R}^n, \|\cdot\|)$ means that the distance of two vectors can be off by at most a factor of D, i.e.

$$\frac{1}{D} d(p,q) \leq \|\iota(p) - \iota(q)\| \leq d(p,q). \quad (2)$$

Such an embedding can be found using semidefinite programming [18] (Section 15.5). Further techniques can be used to reduce the dimension of the representation while keeping a low distortion [1].

D. Distance-preserving Symmetries

In order to combine both approaches of representing mappings, metric space and symmetries, we define a metric on the set of equivalence classes of mappings. Let $[m_i] = \{m \mid m \text{ is equivalent to } m_i\}, i = 1,2$ be two equivalence classes of mappings by considering symmetries. Then, we define the distance

$$d_{\text{sym}}([m_1],[m_2]) := \min_{m\in[m_1], \tilde{m}\in[m_2]} d(m, \tilde{m}).$$

It can be easily shown that if d is a metric, d_{sym} is as well. It follows that $(\{[m] \mid m \in M\}, d_{\text{sym}})$ is also a metric space, and thus, a low-distortion embedding for this representation can be found using the same methods as above.

E. Scalable Embeddings

An embedding $\iota : (\mathcal{M}, d) \hookrightarrow (\mathbb{R}^n\|\cdot\|)$ of a finite metric space $\mathcal{M}$ to the real vector space $(\mathbb{R}^n, \|\cdot\|_p)$ can be described by a real $n \times k$ matrix, where the columns of

the matrix represent the values of $\iota(m_i)$, for $i = 1..k$, where $M = \{m_1, \dots, m_k\}$. While embedding a metric space with 16 elements in a 4-dimensional real space is very manageable, this situation changes drastically if our metric space M has 16^{10} elements and we are encoding it as 1000-dimensional vectors. In that case we need about 10^{15} real numbers to describe ι. It is therefore simple to see why the methods presented do not scale well with a growing mapping space.

To overcome this problem, we can consider the nature of the space and why it grows so rapidly. The reason for this is the exponentially-growing nature of spaces of tuples (or vectors), as can be easily seen by considering the naive vector representation. Instead of trying to find an embedding for the space of tuples $\mathcal{M} = P^{|R|}$, however, we can consider the metric space of the architecture $\mathcal{M} = P$, and define the metric space of mappings of $k = |R|$ tasks as the space $\mathcal{M}^k$, using the same definition for the metric as in Equation (1). In fact, the following theorem ensures that going through this embedding of the base-space we still have the desirable distortion.

Theorem III.1. *Let $\iota : (\mathcal{M}, d) \hookrightarrow (\mathbb{R}^n, \|\cdot\|_p)$ be an embedding with distortion D and define $\iota^k : (\mathcal{M}^k, d_p) \hookrightarrow (\mathbb{R}^{nk}, \|\cdot\|_p)$ as $\iota^k((x_1, \dots, x_k)) = (\iota(x_1), \dots, \iota(x_k))$. Then ι^k is an embedding with distortion of at most D.*

Proof. It is clear why ι^k is an embedding (well-defined and injective), since ι is one. The distortion follows from the homogeneity of the $\|\cdot\|_p$-norm applied to Equation 2. □

Using Theorem III.1 we can devise a very efficient decoding and encoding scheme while ensuring a specific distortion D. We calculate an embedding $\iota : \mathcal{M} \hookrightarrow \mathbb{R}^n$ and store ι and the inverse mapping $\iota^{-1} : \iota(\mathcal{M}) \subset \mathbb{R}^n \to \mathcal{M}$ as look-up tables. To decode ι^d as defined in Theorem III.1, we just divide the vector of $\mathbb{R}^{nk}$ into k vectors in $\mathbb{R}^n$, approximate each to the nearest vector in $\iota(\mathcal{M})$ and apply ι^{-1} component-wise. This reduces the space complexity of the algorithm from $\mathcal{O}(|\mathcal{M}|^k)$ to $\mathcal{O}(|\mathcal{M}|)$, making it scalable.

Note that this approach is at odds with the symmetry reduction, since this separates the components individually, while the symmetries of the problem take into account the relation of the different elements of the mapping. In future work we will explore how to find scalable embeddings that take symmetries into account.

IV. APPLICATIONS

As the landscape of systems and requirements is vast, the objectives on software executing on a system vary accordingly. Depending on these objectives, different representations of the mapping can be ideal for representing a system.

In this section we will present different applications in the context of software synthesis, where for each, different representations of mappings are better suited to achieve the desired objectives. We deal with dynamic runtime systems, compile-time design-space exploration and with a special use case for isolating execution in multi-application scenarios.

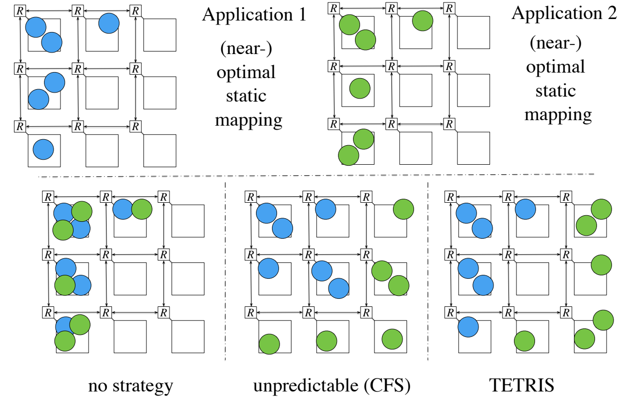


Figure 4. The Idea Behind the TETRiS Runtime System

A. Runtime

At runtime, it is critical for a mapping approach to be quick in making decisions. Therefore, computation times for any mapping-related calculations are central. Representations of mappings that will be used at runtime should thus be very close to the actual thread assignment. This makes the simple vector a very useful runtime representation. For the same reason, a representation based on canonical representatives of mappings is ideal for leveraging symmetries at runtime.

In previous work [10], [4] we have proposed an approach that leverages symmetries at runtime using the simple vector representation with canonical representatives. Figure 4 illustrates the main idea behind TETRiS, our approach. When a new application will be executed on the system, we need a strategy to decide how to map it. If we don't take running applications into account, as depicted by "no strategy" in the figure, we will get very suboptimal results. However, if we instead use a dynamic and unpredictable method, like the Linux scheduler CFS, we will lose the performance properties obtained by calculating a mapping. To solve this, TETRiS takes the canonical representation of the mapping as input and finds a new mapping that is equivalent to it, such that it fits in the unused resources in the system. This has been shown experimentally to improve the predictability of the system, while preserving the properties of the mappings, both in terms of performance and energy consumption, as compared to the Linux scheduler CFS [10].

B. Design-space exploration

Perhaps the largest application for specific mapping representations concerns itself with explicit explorations of the (mapping) design space. It is common of approaches to explore the design-space of mappings by using meta-heuristics. In some easier cases with simple topologies and homogeneous architectures, even brute-force approaches work by formulating the problem as a mixed integer linear programming problem (MILP), or using satisfiability modulo theories (SMT) [17]. However, in more complex scenarios, or when the design-space grows intractably large, other meta-heuristics are common, like genetic algorithms [8], [26], ant-colony optimiza-

tion [7] or even geometric-based approaches like design centering [13].

In general, a metaheuristic is a procedure that iteratively selects points in the design-space in order to optimize some objective(s). The idea is to utilize the structure of the space to explore it more efficiently than an exhaustive search. It is for this reason that most metaheuristic rely on notions of distance or, at least of locality, to search for mappings that are similar to a given one.

In genetic algorithms, for example, the representation is called a *chromosome*, and the mutation and crossover operators are used to find new mappings to explore. The idea of a mutation is for it to be a small perturbation, not changing the mapping significantly. This is where the metric space representations, as described in Section III-C, become very useful, in order to define the distance between two mappings. Thus, a natural way of defining mutations arises, namely, by selecting mappings which are at most at some distance ρ from the original mapping m. Formally, this defines the ball $B_\rho(m) := \{m' \in M \mid d(m, m') < \rho\}$. This can be seen as a way of generalizing the methods from [26].

Another example is the meta-heuristic of design-centering, as in [13]. In it, a heuristic is leveraged that is usually utilized to find parameters in continuous spaces (e.g. values like resistance in circuit design). Using this, closeness of mappings is defined and the method is used to find mappings that are robust to small changes which could occur unpredictably at runtime. Central to this approach is using the geometry of the space and a design center is defined as a point m, such that there exists a ball $B_\rho(m)$ of maximal radius ρ where all points are feasible, or a fixed percentage thereof. Here, feasibility refers to some design objective, e.g. a runtime below a specific (soft) real-time threshold. This example showcases one of a large family of heuristics which are defined using geometric properties of the design space M, i.e. where it is assumed that $M \subset \mathbb{R}^n$ for some n. It is clear how these approaches benefit from a low-distortion embedding, as described in Section III-C.

C. Visualization

Another useful application of having structured representations of the design space is that they allow researchers to visualize the design space. While this would not necessarily be directly relevant during the execution of a DSE phase in an optimizing compiler, it can be extremely useful for researchers and engineers designing such a compiler. In Section V we will describe and show a concrete example of such a visualization.

D. Isolating Applications

Focusing on the representations of mappings makes it easier to concentrate on new properties which might be very difficult to capture otherwise. As an example, consider the two mappings depicted in Figure 5. The figure shows two different multi-application mappings of both applications to a $4{\times}4$ mesh network-on-chip topology. Both mappings use exactly the same PEs of the architecture. In fact, if we ignore contention, both mappings should have the exact same behavior: a careful

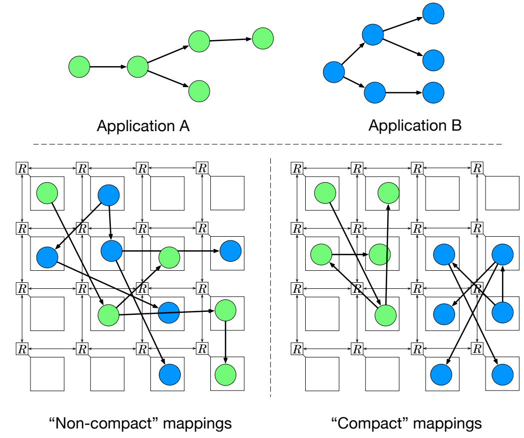


Figure 5. An Intuitive Notion of Compact Mappings

inspection reveals that the distances —in terms of number of hops— between any two communicating tasks in any of the applications are all exactly the same in both mappings. However, it is intuitively obvious that the mapping on the right of the figure is preferable: by being more "compact", the mapping on the right will isolate communication within both applications. This helps to avoid contention and provides increased security. Additionally, it is intuitively simpler to combine single-application mappings like those on the right, since they are less "fragmented".

While intuitive, it is not trivial to define this notion of compactness formally, in a way that an algorithm might try to search for such mappings. In cases where the topology does not have a very straightforward geometric interpretation, this becomes even more complicated. The metric space representation from Section III-C allows us to actually define compactness in a meaningful way, even in cases where the topology is complex.

Mathematical Definition of Compactness

Given a mapping $m : R \to P$ to an architecture with topology graph $T = (P, E)$, we define the *topology graph of the mapping* $T(m)$ to be the subgraph of T that is induced by the image of m on P. This means that $T(m) := (m(R), \{(p, q) \in E \mid p, q \in m(R)\})$, where $m(R) = \{m(r) \mid r \in R\} \subseteq P$.

This induced topology subgraph shows the topology of the mapping. It captures its shape, in a sense. This is what we want to keep compact. For this, we define the *compactness* C of the mapping m as follows: $C(m) := \sum_{\{p,q\} \subset m(\{PEs\}), p \neq q} d_T(p, q)$. Here, d_T denotes the graph metric, just like the one used for defining the shortest path problem. The compactness just adds up the graph metrics of the edges of the induced topology subgraph. Note that we do this in the graph metric of the full topology graph (since the paths that matter are those in the whole architecture, not only in the mapping). A mapping m is *maximally compact* if its compactness is minimal among all subgraphs of T of the same size as $T(m)$.

It can be shown that this concept has some desirable properties. Maximally compact mappings in this sense are not

only connected, they also do not have "holes", and when PEs are missing they are always on the "corners" of the mapping. This makes isolation and composability easier with these kinds of mappings. However, defining and proving this in detail is beyond the scope of this paper.

V. Implementation

In order to explore the applications described in the previous section, we use a python based framework called *pyKPN*[1]. The framework's core component provides classes that model dataflow applications, topology and architecture graphs, as well as mappings. Other components within this framework operate on these common models. For instance, the mapper component implements mapping algorithms that take one or multiple dataflow applications as well as an architecture graph, and produce a mapping.

A. Simulation

A central component of pyKPN is the simulator. It models the execution of a dataflow application on a given architecture implementing a given mapping and provides a performance estimate. The simulator operates on a high level of abstraction and is based on the SimPy module for discrete event simulation [19], [25].

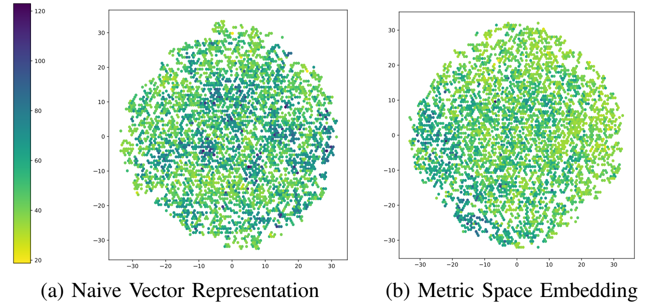
The simulator models the execution of dataflow application by replaying previously recorded *process traces*. A process trace represents one execution of a dataflow node. It divides the execution into multiple segments where each segment denotes a phase of computation between communication events. Here, a communication event is the production or consumption of a token on one of the node's dataflow channels. Traces can be recorded on a host machine or the target device using an instrumented implementation of the dataflow application, that records each communication event. We used the tools from SLX [22] to obtain these traces and estimations.

The simulator focuses on modeling the communication between dataflow nodes. Therefore, it simply models computation as static delays. The length of the processing delay is estimated for the target architecture based on the segment length in the process trace. To correctly simulate the execution of multiple processes on the same processor, the simulator also provides models for schedulers that manage process execution.

Communication events are modeled in more detail. To correctly implement the Kahn-MacQueen blocking semantics [14], the simulator keeps track of the number of tokens available on each dataflow channel. The delay for producing and consuming tokens is computed based on the path taken through the architecture graph. Communication costs may be modeled statically or dynamically based on the system state, e.g., to comprise contention.

B. Visualization

Visually representing real-valued vector spaces with dimensions higher than 2, or perhaps 3, is extremely difficult to impossible in a two-dimensional piece of paper. The embeddings presented in this paper will have a number of dimensions higher than 3 in all but the most trivial cases. However, in order to aid researchers and give an intuition, we can leverage an embedding that is specifically designed to make multi-dimensional spaces visually intuitive. It works by making multi-dimensional spaces two-dimensional in a way that captures some structure and allows for an intuitive inspection of the space. The t-SNE embedding [16] has precisely this goal, which it achieves not by having a low distortion (which is impossible in general), but instead, by striving only to maintain large distances large, and small distances small in the embedding.



(a) Naive Vector Representation (b) Metric Space Embedding

Figure 6. Visualizations of the Design Space using t-SNE

Figure 6 shows the visualization of the mapping space of an application to the Exynos architecture depicted in Figure 1a. The application has 8 tasks, and thus, an 8-dimensional design space. We used the simulator described above to simulate 10000 randomly-generated mappings. In order to visualize them, we used t-SNE as described above (see [16] for more details), with the color-coded labels being the simulated execution time. Note that the algorithm uses randomness, and the axes do not have a meaning, other than points being close in the image means that they are close in the design-space. In the image we show side-by-side both, the naive vector representation and a metric-space low-distortion embedding representation. A careful inspection of the figure shows that the space seems to be more structured in the metric-space representation, having more homogeneity in the clusters of mappings with similar execution times, whereas the naive vector representation yields a more chaotic image.

VI. Related Work

As discussed in Sections II and Section III, the mapping problem is the subject of an extensive amount of research, with several different use-cases, objectives and specialized algorithms. However, this work does not concern itself with any mapping method in particular, but with the way mappings are represented within those methods. This is almost never explicitly addressed in the current literature. Most work either omits describing the representation used, or uses a naive vector representation similar to the one presented in Section III. We have discussed some related work to the presented approaches throughout the paper, wherever relevant.

[1]The camera-ready version will have a link to a github page here.

Some work has, however, concerned itself more directly with the representation. Our own previous work only considers some aspects [11], in this case, particularly the symmetries. The closest work dealing with this is from the authors of [26], which consider representations in a more systematic way, but for limited class of architectures. In particular, their approach only considers homogeneous systems. This work can be seen as a generalization of their methods. Finally, the authors of [21] also consider only a specific representation, symmetries, and also only do so for a limited class of architectures. They explore the mapping space using a symmetry-reducing representation and show how this yields significantly better results.

VII. Conclusion

In this paper we have discussed the mapping problem in software synthesis, and made a case for focusing on the representations of mappings. We showed several representations in a way that is not known from literature, by considering a metric space structure on the mapping space, taking symmetries of the problem into account, and finding efficiently-usable low-distortion embeddings into real spaces, which allows to use large classes of optimization algorithms from literature. We have also shown how these mappings can be useful in a wide variety of different applications within the software synthesis realm.

While there are certainly very useful representations we have not considered in this paper, we believe a stronger focus on the representations in general is indeed imperative to solve the problem efficiently in the future. The methods suggested in this paper should be a start, allowing for fundamentally new ways of looking at these variants of the mapping problem through the lens of representations. More important than improving a particular method, the focus in representation should give new and better abstractions, providing a new language upon which to build better heuristics to solve the mapping problem, in all its diverse incarnations and use-cases.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed). We thank Silexica (www.silexica.com) for making their embedded multicore software development tool available to us.

References

[1] Ailon, N., and Chazelle, B. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing* (2006), ACM, pp. 557–563.

[2] Bhattacharyya, S. S., Murthy, P. K., and Lee, E. A. *Software synthesis from dataflow graphs*, vol. 360. Springer Science & Business Media, 2012.

[3] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing 37*, 1 (1996), 55–69.

[4] Castrillon, J., et al. A hardware/software stack for heterogeneous systems. *IEEE Transactions on Multi-Scale Computing Systems* (Nov. 2017).

[5] Castrillon, J., and Leupers, R. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Springer, 2014.

[6] de Dinechin, B. D., Ayrignac, R., Beaucamps, P.-E., Couvert, P., Ganne, B., de Massas, P. G., Jacquet, F., Jones, S., Chaisemartin, N. M., Riss, F., et al. A clustered manycore processor architecture for embedded and accelerated applications. In *HPEC* (2013), pp. 1–6.

[7] Dorigo, M., Birattari, M., Blum, C., Clerc, M., Stützle, T., and Winfield, A. *Ant Colony Optimization and Swarm Intelligence: 6th International Conference, ANTS 2008, Brussels, Belgium, September 22-24, 2008, Proceedings*, vol. 5217. Springer, 2008.

[8] Erbas, C., Cerav-Erbas, S., and Pimentel, A. Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design. *IEEE Transactions on Evolutionary Computation 10*, 3 (June 2006), 358 – 374.

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

[10] Goens, A., Khasanov, R., Hähnel, M., Smejkal, T., Härtig, H., and Castrillon, J. Tetris: a multi-application run-time system for predictable execution of static mappings. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES'17)* (June 2017).

[11] Goens, A., Siccha, S., and Castrillon, J. Symmetry in software synthesis. *ArXiv e-prints* (Apr. 2017).

[12] Greenhalgh, P. big.little processing with arm cortex-a15 & cortex-a7. *ARM White paper* (2011), 1–8.

[13] Hempel, G., Goens, A., Asmus, J., Castrillon, J., and Sbalzarini, I. Robust mapping of process networks to many-core systems using bio-inspired design centering. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES'17)* (June 2017).

[14] Kahn, G., and MacQueen, D. Coroutines and networks of parallel processes.

[15] Lee, E., and Messerschmitt, D. Synchronous data flow. *Proceedings of the IEEE 75*, 9 (1987), 1235–1245.

[16] Maaten, L. v. d., and Hinton, G. Visualizing data using t-sne. *Journal of machine learning research 9*, Nov (2008), 2579–2605.

[17] Malik, A., Walker, C., OSullivan, M., and Sinnen, O. Satisfiability modulo theory (smt) formulation for optimal scheduling of task graphs with communication delay. *Computers & Operations Research 89* (2018), 113–126.

[18] Matoušek, J. *Lectures on discrete geometry*, vol. 212. Springer Science & Business Media, 2002.

[19] Mller, K., and Vignaux, T. SimPy: Simulating systems in Python. http://www.onlamp.com/pub/a/python/2003/02/27/simpy.html.

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

[21] Schwarzer, T., Weichslgartner, A., Glass, M., Wildermann, S., Brand, P., and Teich, J. Symmetry-eliminating design space exploration for hybrid application mapping on many-core architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[22] Silexica, S. S. G. SLXMapper, 2018.

[23] Silva, J. P. M., and Sakallah, K. A. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design* (Nov 1996), pp. 220–227.

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

[25] Team SimPy. SimPy 3.0.10 documentation. http://simpy.readthedocs.io/en/latest/contents.html.

[26] Thompson, M., and Pimentel, A. D. Exploiting domain knowledge in system-level mpsoc design space exploration. *Journal of Systems Architecture 59*, 7 (2013), 351–360.