mpsym: Improving Design-Space Exploration of Clustered Manycores with Arbitrary Topologies

Andres Goens, Timo Nicolai, and Jeronimo Castrillon, *Senior Member, IEEE*

***Abstract*—With growing numbers of cores, the memory subsystem of manycore architectures increases in complexity. Many modern multicores are designed in a hierarchical fashion, with multiple clusters of processing elements. However, most algorithms for design-space exploration of resource allocation in multicores do not consider these complex topologies, which results in poor scaling, or worse, non-functioning algorithms. In this paper we present `mpsym`, a C++ library designed to alleviate this problem in an algorithm-agnostic fashion. Using methods from computational group theory, we present domain-specific algorithms to improve design-space exploration in hierarchical architecture topologies. We evaluate `mpsym` on multiple design-space exploration algorithms from literature. Without modifying the algorithm, our methods improve the execution time by a factor up to $8.6\times$ on the E3S benchmark suite for complex, clustered architecture topologies. Similarly, by pruning the design space, our methods consistently improve the result of the exploration. In particular, the results from a simulated annealing heuristic on the Kalray MPPA3 Coolidge topology are over $30\times$ better on average, while requiring less time to explore.**

***Index Terms*—IEEE, IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

With the multi- and manycore revolution, hardware architectures continue to increase their complexity and numbers of cores. A higher number of cores greatly increases peak performance, but it also makes it harder to attain such performance in practice. The memory subsystem becomes central to ensuring performance. Cache coherence does not scale to hundreds or thousands of cores, making shared memory models significantly less viable.

Even with distributed memory, data and computation need to be mapped to hardware resources. Since the number of possible mappings grows exponentially with growing numbers of cores and tasks, reasoning about mappings quickly becomes intractable. Significant advances in the mapping to multicores have been made, but most of them make simplistic assumptions about the target architecture topologies that increasingly cease to hold in modern manycores.

In reality, not all systems are simple bus-based architectures without hierarchy or based on Network-on-Chip (NoC) with regular meshes. Instead, most manycores are built hierarchically, with clusters of cores arranged in a particular fashion. As such, the methods devised for these simpler architectures cease to work for modern manycores with hierarchical structures. The Kalray MPPA3 Cooldige [1] chip, for example, has five clusters connected with a Network-on-Chip architecture. In each cluster, a special-purpose and sixteen general purpose cores share a bus-based subsystem.

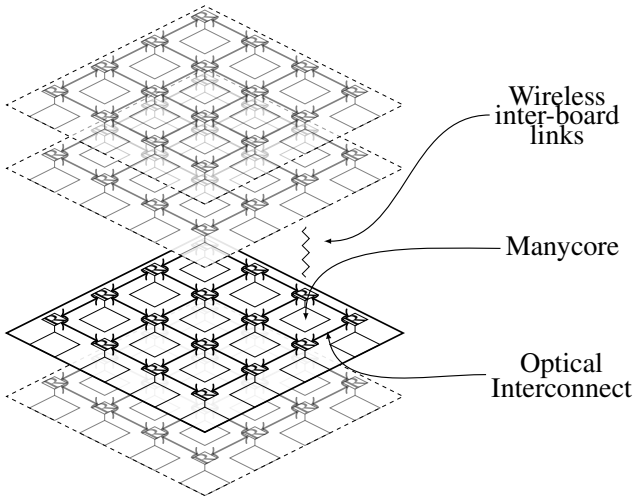

Fig. 1. The HAEC [2] topology with multiple levels of hierarchy.

Future architectures will probably boast even more levels of hierarchy, like the proposed HAEC architecture [2] (cf. Figure 1), where manycores are connected with high-bandwidth optic interconnects on a PCB, and communicate between multiple PCBs with a low-latency wireless interconnect. Improvements in interconnect technology are bound to blur the lines between on-chip and off-chip communication, yielding architectures with several thousands of cores that behave like a single system-on-chip.

How do we reason about the execution of an application spanning multiple clusters in multiple levels of hierarchy? Can we also reason about the heterogeneity, exploiting the accelerators? Multiple methods exist to reason about heterogeneity in architectures, or the memory subsystem in multicores. However, when these two problems are combined, most methods struggle to address both. Moreover, assumptions about the topology of the memory subsystem implicitly or explicitly permeate the models used for programming multicores. These models thus struggle when the topology becomes too complex. We need novel ways to optimize the execution of applications, not only for these architectures, but for even more complex ones in the future. We need a systematic approach to consider heterogeneity and the topology of the network subsystem in design-space exploration (DSE).

In this paper we discuss a mathematical framework precisely for reasoning about these kinds of complex architec-

A. Goens is with the Barkhausen Institut, Dresden, Germany (email: andres.goens@barkhauseninstitut.org). The work for this paper was done in part with the Chair for Compiler Construction, cfaed, TU Dresden, Germany. J. Castrillon and T. Nicolai are with the Chair for Compiler Construction, cfaed, TU Dresden, Germany (emails: timo.nicolai94@gmail.com, jeronimo.castrillon@tu-dresden.de).

ture topologies. A central observation is how hierarchical topologies in hardware architectures exhibit a great amount of symmetry. Figure 2 shows this for a very simple example of mapping two tasks on an ARM big.LITTLE-based 8-core architecture. In the figure, PE_1 to PE_4 are the "little cores", which are slower and more energy efficient, and PE_5 to PE_8 are the faster, albeit more energy-hungry, "big" cores. This is not a realistic problem size, but it allows us to visualize this as a two-dimensional space. The two dimensions of the graph represent the mapping of the two tasks. It is clear from the figure that many mappings are equivalent in terms of their performance, which are the symmetries of the mapping space. Below we show how our method prunes this mapping space by removing such equivalent mappings. The theoretical underpinnings for this symmetry are described in Section III.

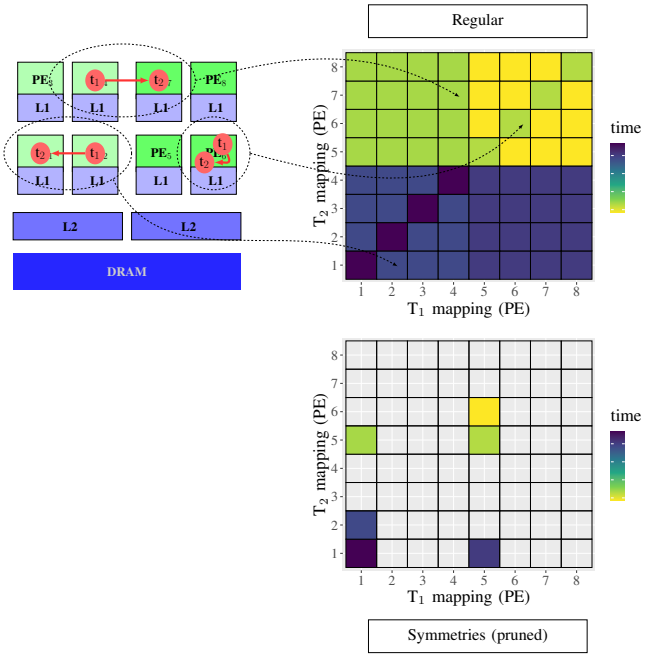

Fig. 2. A simple example of the mapping space and the prunig via symmetries.

We use methods from computational group theory (CGT) to find efficient algorithms that understand these complex structures and their symmetries. In particular, in this paper we present efficient algorithms tailored for DSE of arbitrary hardware architectures (Section IV). Our domain-specific methods leverage the hierarchical nature of architectures to overcome computational limitations of the general cases from CGT. Additionally, we present a probabilistic heuristic which significantly improves the computational overhead of our method, while sacrificing almost no accuracy in practical applications. We have implemented these algorithms in an open-source C++ library, `mpsym`.

We evaluate this framework by using `mpsym` to prune the design space of three meta-heuristics from literature, improving them both in terms of their execution time and the results they produce (Section V). With this evaluation, we show that these methods significantly improve DSE in complex, hierarchical architecture topologies.

II. Related Work

Diverse research methods have been proposed for exploiting symmetries of hardware architectures. The approaches described in [3], [4] also exploit symmetries in a similar context as in this paper, but with other objectives, namely security or run-time scheduling. Many methods for design-space exploration also leverage symmetries [5], [6], [7], [8], [9]. In particular, most of these methods assume a specific topology or family of topologies, like a network-on-chip architecture with a regular mesh structure and no clusters or hierarchy. These methods face limitations when the topology becomes more complex, like star-mesh topologies, or when multiple levels of hierarchy are involved, e.g. in clustered architectures like the Kalray MPPA3 Coolidge. In general, these methods are embedded in full algorithms for design-space exploration (DSE) and depend to different degrees on the specific models used in the algorithm, e.g. the specific topology mentioned.

The methods presented in [10] are not bound to a specific DSE methodology; they can be used to improve basically any DSE for mapping computation to architectures. They also target symmetries, as we do in this paper, but do so with standard algorithms from computational group theory. In particular, their methods do not scale to architectures with more than a few dozen cores or multiple levels of hierarchy. Similarly, the dynamic search-space decomposition presented in [11] is a generalization of [9] that is non-specific to a particular DSE methodology. As such, it is not bound to a concrete topology model: It works for complex topologies and clustered architectures. The basic idea behind the architecture-based optimizations there is to restrict to certain sub-architectures after a pre-exploration step. If this pre-exploration step is ill-suited to deal with the architecture complexity, this approach will not scale to thousands of cores and multiple levels of hierarchy. As such, this work is complementary to the method we present here, since we address specifically the complexity of the topology and hierarchy.

III. Architecture Symmetries

In this section we introduce the techniques for describing and managing symmetries in complex architectures. Our techniques are based on the group-theoretic description of [10], but using domain knowledge of the structure of hardware topologies and the concrete problems to be solved in design-space exploration. We start by reviewing the background on symmetries for improving DSE as described in [10] and then introduce our domain-specific methods for dealing with hierarchical hardware topologies.

A. Architecture Symmetries

Symmetry intuitively refers to agreements in proportions or arrangement, or to the relationship of parts to a whole [12]. The mathematical theory of symmetries aims to capture this intuition by describing symmetry through transformations. Mathematically, a symmetry thus refers to a transformation of an object which preserves its structure.

To describe these mathematically, we need to model the architecture as a labeled multigraph Γ. The nodes of this graph represent the processing elements (PEs) of the architecture and are labeled by their (micro)architectures and frequencies. The edges, on the other hand, represent the possible communication between PEs. They are labelled with communication primitives: an abstraction that models ways of exchanging data in the architecture, like shared memories, caches or direct memory access (DMA). Figure 3 shows an architecture graph Γ for the big.LITTLE architecture (cf. Figure 2).

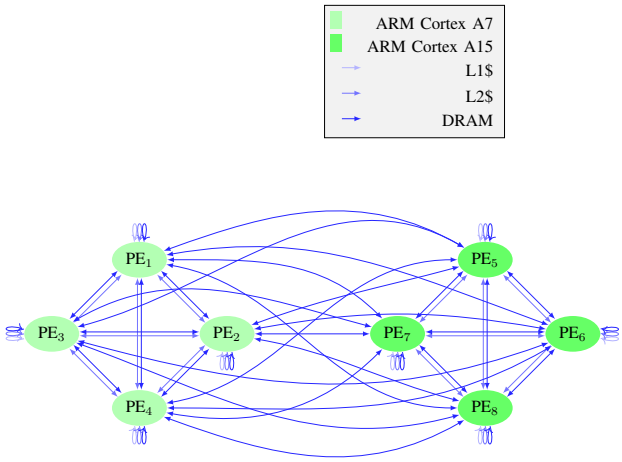

Fig. 3. An example of the architecture graph Γ. Symmetries of the architecture can be formalized as automorphisms of this graph.

With the description of architectures as graphs, we can study their symmetries. Consider the architecture depicted in Figure 4. It has 16 identical general-purpose cores and a secure and management core. They can communicate via a shared secure bus. This figure is in fact based one of the five clusters of the Karlay MPPA3 Coolidge architecture [1]. The figure depicts a transformation as an example of a symmetry. If we change how we name the 16 identical cores, reversing their order, the topology of the architecture remains unchanged.

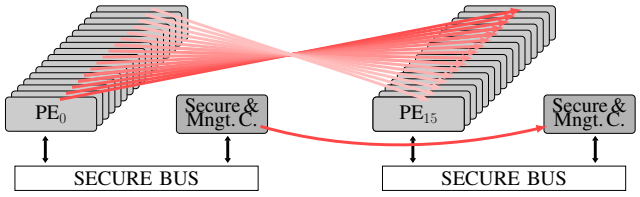

Fig. 4. Symmetries can be described as transformations that preserve the structure.

The transformation from Figure 4 can thus be described as a function from the graph Γ to itself:

$$\varphi : \mathrm{PE}_0 \mapsto \mathrm{PE}_{15}, \mathrm{PE}_1 \mapsto \mathrm{PE}_{14}, \ldots, \mathrm{PE}_{15} \mapsto \mathrm{PE}_0, \\ \mathrm{PE}_{\text{secure \& mgt.}} \mapsto \mathrm{PE}_{\text{secure \& mgt.}}$$

The transformation φ preserves the structure of the architecture. Formally, the edges and all labels of Γ are preserved under φ. This transformation can be undone, since φ is a bijection of the cores. Similarly, if we take a second transformation σ that preserves the structure of the architecture, then

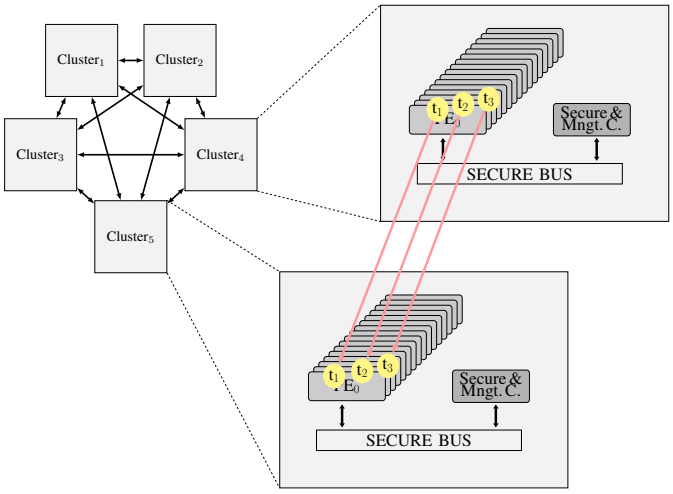

Fig. 5. An example of an action on a mapping to the Kalray MPPA3 Coolidge Architecture. Note that clusters are connected via a NoC, which is not explicitly depicted in this figure.

the composition $\varphi \circ \sigma$ will also preserve the structure of the architecture. Finally, it is clear that the identity transformation $\mathrm{Id}_{\{\mathrm{PE}_0,\ldots,\mathrm{PE}_{15}\}}$, which changes nothing, also preserves the structure of the architecture. Together, these properties define a mathematical structure called a *group*.

In general, structure-preserving bijections like these are called *isomorphism*. In other words, the symmetries of the architecture are the isomorphisms from the architecture graph Γ to itself. Isomorphisms from an object to itself are called *automorphisms*, which we denote by $\mathrm{Aut}(\Gamma)$. For the example of Figure 4, this group is the set of all permutations of the 17 cores (if we count the secure core) which preserve the structure of the architecture. In the case of this simple example, this results in precisely the permutations of the 16 identical cores, leaving the secure core unchanged. In general, the set of all $n!$ permutations on n points is always a group. It is called the *symmetric group* S_n.

The topology of an architecture as modeled by the graph Γ is clearly an abstraction. If we consider the actual floorplan of the chip, the symmetry as described in Figure 4 ceases to hold. This can be important, e.g. when considering thermal effects. Similarly, through process variation the cores are likely not physically identical. From the point of view of software, however, we cannot distinguish the 16 identical cores. As such, we cannot distinguish any mapping of tasks to these cores in terms of e.g. performance or energy efficiency, at least not a priori. This observation, going from a symmetry of the architecture to symmetries in mappings of software tasks to it, is captured by the mathematical concept of an action.

Intuitively, a group action relates the abstract concept of a group G with the symmetries of an object Ω, by specifying how the abstract group elements "act" on objects of Ω. Formally, the (left) action of a group G on a set Ω is defined via a function $\alpha : G \times \Omega \to \Omega$ that respects the group multiplication, i.e. with $\alpha(gh, \omega) = \alpha(g, \alpha(h, \omega))$ and $\alpha(1_G, \omega) = \omega$ for all $g, h \in G, \omega \in \Omega$. Here, $1_G \in G$ denotes the neutral element of G, like the identity function $\mathrm{Id}_{\{\mathrm{PE}_0,\ldots,\mathrm{PE}_{15}\}}$ in the example of Figure 4. In our notation we omit the α for

simplicity and write $\alpha(g, \omega)$ as $g\omega$. The automorphism group of the architecture acts on the set of cores. The automorphism depicted in Figure 4 is an example of this via $\alpha(\varphi, \omega) = \varphi(\omega)$ for an automorphism $\varphi \in \mathrm{Aut}(\Gamma)$ and $\omega \in \Gamma$. In other words, the action of $\mathrm{Aut}(\Gamma)$ on Γ is just function application.

We can consider an automorphism φ of the architecture as a renaming of the PEs, without affecting their types or communication relationship. If we have a mapping of tasks to the architecture, we get a new mapping by renaming the PEs in the mapping according to φ. This new mapping should have the same runtime performance characteristics. More formally, the action of the automorphism group of the architecture also induces an action on mappings. If we consider a mapping as a function $m : T \to P$ from the set of tasks to the set of cores, then an action of G on P induces an action on the set of mappings M via $gm := t \mapsto g\ m(t)$. This means that an automorphism changes the processor mapping of all tasks according to the architectural symmetry.

Consider the full topology of the Karlay MPPA3 Coolidge [1], depicted in Figure 5. This architecture has five identically clusters, each one like the cluster depicted in Figure 4. The five clusters are fully connected via a Network-on-Chip (NoC). This model is a simplified versiod of the actual chip, which contains many additional features like DMA units in each cluster or crypto accelerators. We consider it for the topology. The group of automorphisms of this architecture is the set of the permutations of the 85 cores (again counting the secure cores) which preserve the structure of the communication subsystem. For example, any rearranging of the 16 identical cores in the first cluster (like in Figure 4), or swapping the last two clusters, both preserve the structure of the architecture. We could not, however, change the secure cores of the first and second cluster while leaving the rest unchanged: this changes the communication subsystem, as communication from the secure core to a general purpose core in the same cluster will have a higher latency if we move the secure core to a different cluster.

Figure 5 depicts how the automorphisms of the architecture induce symmetries of mappings via an action, as described above. The figure depicts an example of a mapping of three tasks to the first three cores in the fourth cluster, $m : t_1 \mapsto P_{4,1}, t_2 \mapsto P_{4,2}, t_3 \mapsto P_{4,3}$. The transformation g shown swaps the fourth and fifth clusters. It changes this mapping to $gm : t_1 \mapsto gP_{4,1} = P_{5,1}, t_2 \mapsto gP_{4,2} = P_{5,2}, t_3 \mapsto gP_{4,3} = P_{5,3}$.

B. Hierachical Architectures

A crucial observation in describing architecture topologies is the way they are usually built. Modern manycores are almost universally built in clusters, using multiple levels of hierarchy. We can leverage this fact and describe the topologies of these manycores using particular constructions in group theory. In other words, we combine groups in certain ways to obtain new groups, describing the symmetries of these hierarchical architectures.

Given two groups G, H the direct product $G \times H$ is the group on the Cartesian product $G \times H$ with a component-wise composition, i.e. $(g, h)(g', h') = (gg', hh')$ for all $g, g' \in G, h, h' \in H$. Intuitively, this takes two groups on n and m points and combines them to act on $n+m$ points, where each of the groups acts independently of the other. For example, if we take an architecture that has four identical cores in a single cluster, its symmetry group is S_4, the symmetric group on four points. Consequently, if we have two clusters each with 4 cores of different types, like in the architecture from Figure 3), then the symmetry group of the architecture is $S_4 \times S_4$, the direct product of the symmetry groups on each cluster.

Similarly, there is another construction called the *wreath product*: Let G, H be groups, and further let H be a permutation group on n points. Consider an element in the direct product of n copies of $G, (g_1, \ldots, g_n) \in G^n = G \times \ldots \times G$. We can apply an element $h \in H$ to this element by ${}^h(g_1, \ldots, g_n) = (g_{h1}, \ldots, g_{hn})$, i.e. by permuting the order of the elements in the n-tuple. This defines an action of H on G^n. We can use this action to construct the wreath product $G \wr H$ on the Cartesian product $G^n \times H$, by the multiplication[1]:

$$\begin{aligned}&((g_1, \ldots, g_n), h)((g'_1, \ldots, g'_n), h')\\ &= ((g_1, \ldots, g_n)\,{}^h(g'_1, \ldots, g'_n), hh')\\ &= ((g_1, \ldots, g_n)(g'_{h1}, \ldots, g'_{hn}), hh')\end{aligned}$$

Intuitively, the wreath product works when we have copies of a substructure arranged in a particular larger structure, by applying the transformations at both levels. For example, the symmetry group of the Kalray architecture is $G \wr S_5$, where G is the symmetry group of a single cluster. This can be seen in Figure 5, which shows the two levels of hierarchy separately. The symmetry group H of the clusters is S_5, since they are fully connected. The symmetry group G of a single Cluster is *isomorphic* to S_{16}. This means the two groups have the same structure. However, as modeled here, G is a group on 17 points, corresponding to the 17 cores. Since the secure cores are different to the 16 general purpose cores, the symmetry group G of a single cluster does not permute them. As such, the first group G acts on the smaller structures (the clusters), and the second group $H = S_5$ permutes the clusters.

In general, we can see how there is a correspondence between the constructions we use to build hierarchical hardware architectures and the group constructions presented here. Table I gives an overview of the different constructions. When all cores are interchangeable, like in a cluster with identical cores and homogeneous communication, the symmetric group S_n can be used to describe their symmetries. For combining distinct resources, like the heterogeneous clusters in an ARM big.LITTLE platform (cf. Figure 3), we use the direct product to combine the symmetries of each sub-structure. For some structures, like a Network on Chip (NoC) with a complex topology[2], we need to calculate their symmetry group by solving the graph isomorphim problem. This can be done efficiently in practice [13]. Finally, to compose the symmetries of two different levels of hierarchies, we use the wreath product. This is a special case of the composition of distinct elements/clusters, since there the higher abstraction level in the

[1]Both this and the direct product are special cases of the semidirect product $G \rtimes_\varphi H$ for $\varphi : H \to \mathrm{Aut}(G)$

[2]assuming the differences from latencies in the NoC are not negligible

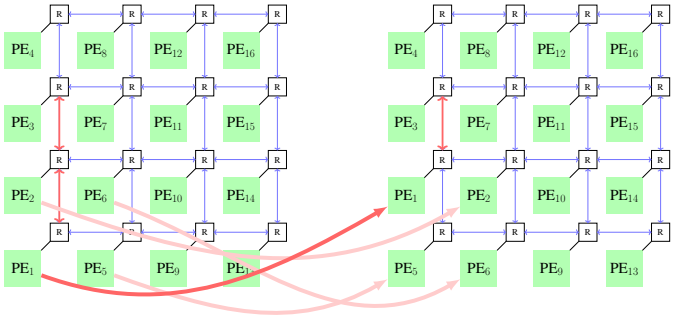

Fig. 6. An example of a partial symmetry in a NoC-based architecture.

hierarchy has no symmetry. This is consistent with Table I, as the direct product can be seen as special case of the wreath product for a trivial group acting on two points.

C. Limitations

The symmetries described in this way are an approximation, and as such, have some limitations. As discussed above, there are many properties of a real system that break these symmetries. Process variation might make nominally identical cores behave differently. Similarly, thermal effects from the actual physical placement in the architecture might make equivalent mappings under this formalism non-equivalent. This can result in dynamic frequency scaling, which also changes the behavior of cores. If the dynamic scaling is not managed identically on the different cores, as might be the case, e.g. because of thermal effects, our symmetries become invalid. Similarly, the execution of the application depends on access to peripherials and the latency to this access is not homogeneous, this also breaks the symmetries as described here.

These limitations can be summarized as the level of abstraction at which the models operate. Our symmetries are valid at the system level, at the level at which our simulations work. Since the effects outlined above are commonly not considered in the simulation either, the results of a system-level simulation will be identical for equivalent mappings.

A special case that warrants discussion is the on-chip interconnect. Bus-based architectures might exhibit different latencies for the different cores depending on the logic inside the interconnect. This is normally not considered in system-level simulations.

On the other hand, Network on Chip (NoC)-based architectures sometimes are explicitly considered in simulations. The latency can sometimes be predictably different for different routes in the NoC. Our methods model this as was described above (cf. Table I).

The formalism as described here is not well-suited to describe the symmetries within a NoC. Consider the architecture depicted in Figure 6, a regular 4×4-mesh NoC. The transformation depicted in the figure rotates the cores in the lower-left 2×2 sub-mesh. This is transformation does not preserve the structure of the NoC (is not an automorphism of Γ), since it breaks the labels of the edges. This is depicted in the figure by comparing the distance between PE_1 and PE_3. As marked in the figure, the distance increases with this transformation. However, for a mapping that only uses the lower-left sub-architecture, describing this symmetry will find additional equivalent mappings. Additionally, the architecture graph as defined here does not consider the concrete route in the NoC. This not only disregards contention, it also fails to model more complex routing strategies, like dynamic routing methods or wormhole switching. This type of symmetry can be modeled with a generalization of group called inverse semigroups [12], [10], modeling the transformation in Figure 6 as a partial function. In this context, a different graph model has to be used to properly describe the architecture topology, such that it includes the routes.

In this paper we choose to ignore partial symmetries and model only global ones, as groups. While this is a limitation of our model, we argue it is not decisive. This decision is based on the observation that the differences in the latency between different routes in a NoC are usually just a handful of cycles, and thus not very large in comparison to the execution times or the overall communication costs. Approximating a NoC as being a uniform system of communication when finding mappings is not common in practice. It might not be well-suited for all use-cases, like very tightly constrained hard real-time systems. However, for best-effort scenarios or even some soft or firm real-time systems, we argue it is a good approximation. It is the hierarchy in the memory subsystem that dominates communication costs, not the routing with a NoC. We have confirmed this approximation to be good in preliminary results, where we showed the geometry of the mapping in the NoC is mostly irrelevant for mappings in practice [14]. Future work should continue to investigate this issue. For now, the issue of modeling NoCs should still be considered a potential limitation of our approach.

IV. The mpsym Library

In this section we describe the implementation of `mpsym`, an open-soruce[3] C++ library for accelerating DSE for complex architecture topologies, using the methods described in Section III. The core design principle of `mpsym` is to use domain-specific knowledge for modeling complex architecture topologies (cf. Table I), while leveraging the machinery of CGT. Figure 7 shows the general flow of using `mpsym`. It works as a light-weight library that can be readily added to any DSE algorithm, improving it by leveraging the symmetries of the problem. By changing the mapping representation to factor out the symmetries [15], `mpsym` works without having to modify the DSE algorithms at all, only changing the underlying design space (cf. Section IV-B). `mpsym` implements multiple algorithms from CGT as well as our domain-specific extensions. This section explains the algorithmic design of `mpsym`.

A. The Schreier-Sims algorithm

A very important concept in CGT is that of a *generating set*. For a group G, a set $X \subseteq G$ is a generating set of G if every element $g \in G$ can be written as a word in the elements[4] of X,

[3] https://github.com/tud-ccc/mpsym

[4] in general, we also need the inverses, but for finite groups as is the case here, the elements suffice

Hardware Architecture	Group Theory
Homogeneous substructure (n identical PEs or clusters)	Symmetric Group S_n
Distinct elements/clusters	Direct product $G_1 \times \ldots \times G_n$
NoC Connection with topology graph Γ (identical elements)	Automorphism group of Γ $\mathrm{Aut}(\Gamma)$
Hierarchical composition	Wreath product $G \wr H$

TABLE I
CORRESPONDANCE OF ARCHITECTURE AND GROUP-THEORETIC CONSTRUCTIONS.

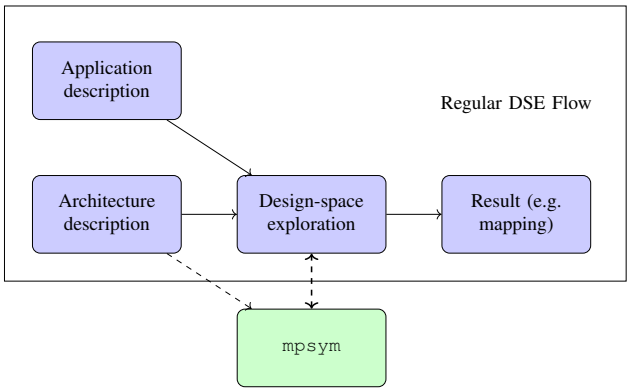

Fig. 7. The mpsym Software Flow

i.e. $x_1, \ldots, x_n = g$ for $x_1, \ldots, x_n \in X$ and $n \in \mathbb{N}$. We can think of a generating set as a lazy data structure that represent the group G. Not every generating set is equally good as a data structure, however, and the Schreier-Sims algorithm can be used for finding particularly efficient generating sets.

The Schreier-Sims algorithm is indeed a central pillar of CGT [16]. Given a generating set X for a permutation group $G \subseteq S_n$, it computes a special data structure called *base and strong generating set* (BSGS) which is a special generating set and represents G. The details of this data structure and the algorithm are beyond the scope of this paper. Important is, however, how it makes several computations required by mpsym's DSE approach feasible. In particular, given a BSGS for G, it is possible to efficiently determine $|G|$, uniquely enumerate all $g \in G$ and to check whether $s \in G$ for some $s \in S_n$.

In practice, there are variants of the Schreier-Sims algorithm that can be significantly faster in some cases. The random Schreier-Sims algorithm, for example, is a very fast Monte Carlo algorithm which can fail with a given probability, controllable via a parameter. There exist more advanced algorithms to guarantee the correctness of its result, such as the *Todd-Coxeter* Schreier-Sims algorithm. It is not obvious in general which algorithm is best-suited to construct a BSGS from a given generating set. Computer-algebra systems, like GAP [17], use heuristics to decide which algorithm or combination of algorithms to apply. The mpsym library implements the deterministic and random variants. A reasonable "default" approach to constructing a BSGS for a large group is to run the random Schreier-Sims algorithm followed by a correction step, e.g. running the deterministic Schreier-Sims on the resulting BSGS, which is the naive approach employed by mpsym.

B. Finding canonical forms

For several use-cases in DSE, e.g. optimizing for performance or energy consumption, two mappings that are equivalent via symmetries can be considered as identical. To leverage this, we want to "factor out" the symmetries. Formally, we want to explore the sets of orbits $M \backslash G$, instead of the mapping space M. An orbit is the set of all mappings that are equivalent via symmetries. For $m \in M$:

$$\begin{aligned} Gm &= \{m' \in M \mid \exists g \in G : gm = m'\} \\ &= \{gm \mid g \in G\} \in M \setminus G \end{aligned}$$

To work with orbits, we consider a canonical representative of every orbit, i.e. an element $\hat{m} \in Gm$ for every $Gm \in M \setminus G$. We sort the cores P (e.g. alphabetically by name) and tasks T, and define an order on the set of mappings M by the lexicographic ordering[5] on the tuples $(p_1, \ldots, p_{|T|})$, where task t_i is mapped to the processor $p_i \in P$.

This factorization of orbits is precisely what Figure 2 from the introduction depicts. In the figure, all mappings with the same color are exactly the mappings in each orbit. The elements that remain in the pruned space in the lower part of the figure are the canonical representatives, the lexicographic-minimal elements of each orbit.

mpsym can find these representatives by completely enumerating Gm. In practice this is achieved either by enumerating all $g \in G$ or by using the orbit algorithm described in Section IV-C. The latter is usually more efficient when $|Gm| \ll |G|$. In mpsym, we denote the first of these two approarches as `iterate` and the second one as `orbit`. Note that `iterate` refers to iteration on the group itself, whereas the iteration in `orbit` is based on the mapping space that the group acts on.

Alternatively, we can use some form of local search, e.g. by initially setting $m^{(1)} = m$ and then, given a generating set X for G, iteratively considering $sm^{(i)}$ for all $s \in X$. We can then set $m^{(i+1)} = sm^{(i)}$ if $sm^{(i)} < m^{(i)}$, until we reach a (local) minimum. This approach can be enhanced by employing heuristic techniques such as simulated annealing. It can potentially be much faster than complete enumeration, but this is not guaranteed to yield the actually lex-minimal element of the orbit $\hat{m}$. However, if we want to consider $G \setminus M$ for pruning in DSE, it is not very problematic that the local-search heuristic can fail. In the worst-case, the design space is not pruned as much as it could be. In mpsym, this approach is denoted by `local search`.

Using either complete enumeration or local search, mpsym is furthermore able to determine $\hat{m}$ especially well for certain

[5]This is an arbitrary choice, any ordering that is easy to compute should work as well.

decomposable automorphism groups, see Section IV-D. Both local search and optimizations for decomposable automorphism groups were first used in [18] in the context of model checking. Leveraging this decomposition is what makes our domain-specific approach more efficient.

C. Finding orbits

Other uses in the context of mappings to manycores require to explore the orbits themselves, like in hybrid mapping approaches [4]. For this, we want to lazily calculate elements of the orbit $Gm = \{gm \mid g \in G\}$. We do this by applying all generators $s \in X$ of the generating set X obtained by the Schreier-Sims algorithm to the mapping m. By iteratively doing this on all the points obtained this way, we get the full orbit Gm. This is a standard algorithm called the orbit algorithm.

We also expose this as a lazy enumeration in `mpsym` by only applying the generators when additional elements of the orbit are required, and keeping a hash function of the elements found, to only return new elements obtained. Note however that completely enumerating orbits can often be impracticable with respect to available memory. This is because $|Gm| \leq |G|$ can be very large (e.g. for S_n up to $n!$). Additionally, the chosen hash function must be perfect to ensure correctness, meaning that we usually require several bytes to represent each orbit element's hash.

D. Wreath-Product Decomposition

Finally, a central optimization of our domain-specific approach is the use of wreath products to describe hierarchical architecture topologies, like the HAEC or Kalray MPPA3 Coolidge architectures.

If we have an automorphism group given as a wreath product $G \wr H$, we can accelerate the relevant algorithms for finding canonical representatives for some mapping m. This is possible by formulating an equivalent problem in which we need to find canonical representatives for a number of groups $\sigma_1(G), \sigma_2(G), \ldots, \sigma_{\deg(H)}(G)$ and $\sigma(H)$ which are trivially constructible from G and H. This is usually much less computationally expensive than directly determining them from $G \wr H$, especially when $|G \wr H|$ is very large.

This algorithm, as well as an algorithm that can automatically decompose groups into wreath products (where possible) are described in [18]. While `mpsym` implements the latter algorithm, usually it is not needed, as we can determine the decomposition directly from the construction correspondence described in Table I.

E. Architecture Description Language

As explained above, we leverage the correspondence from Table I to generate an efficient data structure and optimized algorithms for this case. To facilitate this correspondence, we have developed a domain-specific language in Lua. The language can be used to easily describe topologies and derive their symmetries automatically from this description. An example for the HAEC architecture presented in Figure 1 is given in Listing 1. It showcases our wreath product construction, which works by using two structures, `proto` and `super`. The `proto` structure corresponds to the group G in $G \wr H$, while H is the symmetry group of the `super` structure. In the listing, the `proto` structure are the optical interconnects of each of the four layers in the HAEC topology (cf. Figure 1). The functions `mpsym.identical_processors` and `mpsym.grid_channels` are used to describe this NoC mesh with 16 identical processors. Similarly, the `super` graph represents the higher level of hierachy, where each of the elements represented is one of the four clusters or layer in the architecture. The functions used here like `identical_processors`, `identical_clusters` and `ArchUniformSuperGraph` implement the constructions described in Table I.

Listing 1. HAEC.lua - Topology Description

```
local mpsym = require 'mpsym'

local super_graph_clusters =
  mpsym.identical_clusters(4, 'SoC')
local super_graph_channels =
  mpsym.linear_channels(super_graph_clusters,
                        'wireless')

local proto_processors =
  mpsym.identical_processors(16, 'P')
local proto_channels =
  mpsym.grid_channels(proto_processors,
                      'optical')

return mpsym.ArchUniformSuperGraph:create{
  super_graph = mpsym.ArchGraph:create{
    directed = false,
    clusters = super_graph_clusters,
    channels = super_graph_channels
  },
  proto = mpsym.ArchGraph:create{
    directed = false,
    processors = proto_processors,
    channels = proto_channels
  }
}
```

These architecture-description scripts can be parsed via `mpsym`'s C++ or Python interface. `mpsym` is then able to determine a topolgy's automorphism group as well as canonical representatives for arbitrary mappings to that topology. Internally, `mpsym` makes use of the well known program *nauty* [13] to determine generating sets for the automorphism groups of architecture graphs. Listing 2 showcases how to parse and utilize the architecture description in `mpsym` using the python interface `pympsym`. The call `from_lua_file` parses the file from Listing 1 and initalizes mpsym with the symmetries of the HAEC architecture. The next call, `r = ag.representative` calculates the canonical representative for the mapping `(12,13,14,15)`. This mapping is equivalent to `(0,1,2,3)`, which is the lexicographical minimal mapping in its orbit. Thus, `r` will have this (canonical) mapping stored after the call.

The code in Listing 2 showcases how our approach can be used in an algorithm-agnostic fashion, in any meta-heuristic.

Calling `ag.representative` every time an operation will be made with a mapping effectively prunes the design space. This can thus be used in any meta-heuristic. Alternatively, by using `ag.representative` to generate a key for a simulation cache, we can create a symmetry-aware cache that reduces the number of simulations in a mapping algorithm without changing its behavior nor results.

Listing 2. HAEC.py - Use Topology Description to Find Canonical Representatives

```
import mpsym

ag = pympsym.ArchGraphSystem.\
    from_lua_file("haec.lua")

r = ag.representative((12, 13, 14, 15))
#r = (0, 1, 2, 3)
```

F. Limitations

We have argued how `mpsym` can be used for DSE in a variety of settings. The DSE settings in which `mpsym` can be used is still limited by some general assumptions of our method. As discussed in Section III, our symmetries work at a system-level. In particular, for a DSE approach to benefit from `mpsym`, it has to operate at this level of abstraction.

To interface with the library, we need a model of the architecture that can produce the architecture graph Γ. While this is not a strong limitation, it can represent a moderate implementation overhead to use our library.

Finally, we have discussed how our methods in `mpsym` can work with any meta-heuristic. While this is technically correct, without any assumptions on the meta-heuristic, pruning the design space might not be beneficial. In particular, `mpsym` and symmetries are useful when the meta-heuristic works by systematically exploring the design space e.g. by iteratively changing the mapping in some way and evaluating it to guide the search. Since for most practical instances the design space is extremely large, even after pruning, a uniformly random sample of the design space will behave identically after the pruning. In other words, without structure in the search, symmetries might not make a large difference, as we shall see in the evaluation in Section V.

V. Evaluation

In this section we evaluate our `mpsym` library. We do this by first demonstrating that the performance of the basic BSGS construction algorithms utilized by `mpsym` is comparable to those implemented by the state-of-the-art computer-algebra system GAP [17]. We then demonstrate `mpsym`'s ability to perform fast mapping normalization, especially for hierarchical architectures. Finally, we use `mpsym` to improve the execution time and the quality of the design points found during an end-to-end DSE task.

A. Comparison to GAP

`mpsym` implements a small set of well known CGT algorithms that are relevant to the DSE task, as described in Section IV. These algorithms are also implemented by well-established computer-algebra systems such as GAP. We thus investigate whether the performance of `mpsym` can measure up to that of GAP with respect to BSGS construction.

We emphasize that GAP is not a DSE tool, but rather, a general purpose computer-algebra system with a special focus on CGT. It combines an interpreted programming language and a large ecosystem of libraries. It would have been entirely possible to implement `mpsym` in the GAP language, making use of its various highly optimized CGT functions. However, this approach would have had a number of drawbacks. Interfacing with GAP code is cumbersome. While mechanisms like OS pipes can be used for communicating with other programs (e.g. a multicore compiler), they require the full GAP interpreter to be executing. Furthermore, running interpreted GAP code can result in long startup times. In our setup, just starting the GAP interpreter took around a second (this overhead is not included in our evaluations).

We implemented `mpsym` from the ground up in C++, with bindings to Python. We believe that this will result in significantly easier adoption and reuse of our code. In this context, we only aim to achieve a performance that is comparable to that of GAP. The latter employs more sophisticated BSGS construction heuristics and algorithms. Adding these to `mpsym`, while feasible, would not have had a significant impact on the findings in this work.

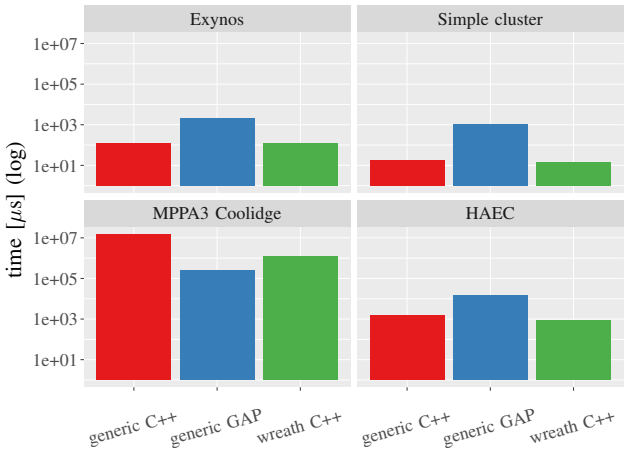

Fig. 8. Comparison between GAP- and C++ (`mpsym`)-based implementations (Schreier-Sims).

Figure 8 shows a comparison of BSGS construction execution times for four different architectures: Exynos (cf. Figure 2), Simple cluster, MPPA3 Coolidge (cf. Figure 5) and HAEC (cf. Figure 1). Exynos refers to the Samsung Exynos 5 and 7 families with an octacore big.LITTLE (4+4) architecture. Its topology is the only non-hierarchical one among these four. Simple Cluster is an architecture comprised of two identical clusters, each with two identical cores. Its topology is the smallest non-trivial hierarchical example that benefits from our novel methods. For the variants, generic refers to generic group algorithms as presented in [10], while wreath refers to the domain-specific decomposition using wreath products. Note that while GAP implements wreath products, we did not re-implement `mpsym` in GAP, which is why we omit the wreath GAP variant.

We repeat the construction 100 times and report the arithmetic means. Note that we omit the variance since it is small enough that it would not be visible in the figure. All experiments were executed on an Intel© Core™ i7-9700T CPU @ 2.00GHz.

The deterministic Schreier-Sims algorithm for wreath-product decompositions, as implemented in `mpsym`, is generally the fastest variant in the evaluation, except for the Coolidge topology. Constructing a BSGS for this particular topology is computationally expensive due to the large order of the corresponding automorphism group ($> 10^{214}$). Here, GAP outperforms `mpsym`. This is likely because of the more advanced heuristics and BSGS construction algorithm variants it employs.

There exist more sophisticated BSGS correction algorithms such as the Todd-Coxeter-Schreier-Sims algorithm [19]. Through the use of such algorithms, GAP can efficiently construct a BSGS for groups of very large order explaining its orders of magnitude better performance for the Coolidge topology. Currently, `mpsym` does not implement any such algorithm. However, we argue this is only a minor limitation. Figure 8 only serves to show that BSGS construction as implemented in `mpsym` is reasonably fast. For large hierarchical topologies like Coolidge or HAEC, constructing a full BSGS is not necessary for DSE as described in section IV, such that the overall overhead incurred by integrating `mpsym` into a DSE flow remains small as we will see in the following. Finding automorphism group generators using Nauty and subsequent BSGS construction are only necessary once per topology. Because of this, `mpsym` implements a mechanism for precomputing BSGS(s) and storing them in a JSON file that can later be parsed prior to DSE.

B. Mapping Normalization

We investigate the execution times of the mapping normalization with the three approaches presented in Section IV for the same four architectures. Since the difficulty of normalizing a mapping depends on its size (i.e. number of tasks) we compare the execution time for mappings of increasing sizes ($1, 2, 4, 8, 16$ and 32). We normalize 100 randomly generated mappings for each of those mapping sizes. Figure 9 shows the average execution times with standard deviation estimators depicted as error bars.

Note that while local search is not guaranteed to yield correct results, its accuracy was perfect in our tests. This is likely because all relevant automorphism groups are relatively small, in part thanks to wreath product decomposition. We applied the latter whenever possible, i.e. for all architectures except Exynos. For large hierarchical architectures, performing mapping normalization can otherwise be outright infeasible, i.e. for the Coolidge architecture where normalizing just a single mapping is otherwise not possible even with several minutes of computing time.

Notice that even for the large Coolidge architecture, normalization is very fast. In particular, it is only slightly slower than normalization for the small Exynos architecture when using the `iterate` approach and even faster for larger

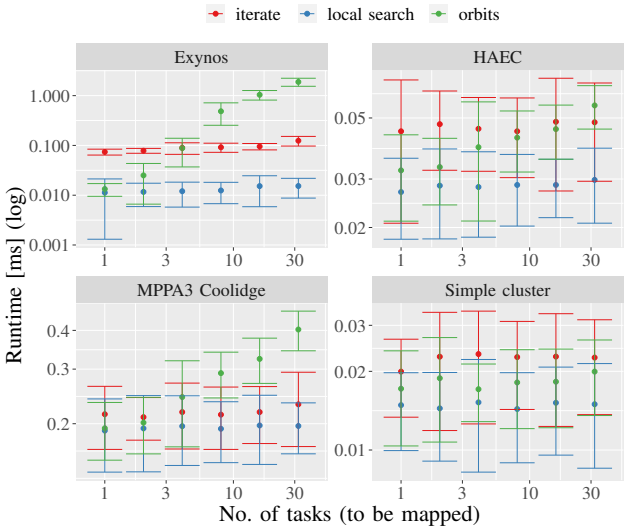

Fig. 9. Comparison between the three mapping normalization heuristics.

mappings when using the `orbit` algorithm. For this particular architecture, decomposition is particularly effective because its clusters exhibit a topology whose corresponding automorphism group is isomorphic to a symmetric group, for which normalization is trivial. This demonstrates that our approach scales very well to large hierarchical architectures. For the three hierarchical architectures all normalization algorithms perform similarly, with orbit enumeration tending to slow down with increasing mapping size and local search generally being the fastest option. We use local search for the remainder of this evaluation.

C. Design Space Exploration

Having shown that `mpsym` is on-par with an equivalent implementation utilizing state-of-the-art CGT algorithm implementations, we will now examine how it fares in its intended use. For this, we evaluate our methods on an end-to-end DSE task. For our evaluation we consider the Embedded System Synthesis Benchmark Suite [20] (E3S). This benchmark suite consists of 20 benchmarks with up to 9 task each, coming from multiple embedded domains: automotive/industrial, telecommunications, networking, consumer and office automation. We use a methodology similar to [8], where we consider different architecture topologies with the resources described in the E3S benchmarks. For simulation and DSE, we employed mocasin [21], an open-source python DSE framework with a discrete-event simulator, and use `mpsym`'s Python interface. With this integration, `mpsym` can automatically derive the symmetries from any architecture format supported by mocasin, like the ones based on IEEE Standard 2804-2019.

1) Genetic Algorithms: We start by reproducing the evaluation from [10] by applying our method to a genetic algorithm. The genetic algorithm follows the general approach used by Sesame [22]. The basic idea of this evaluation method is to replace a simulation cache with a symmetry-aware simulation cache. Two mappings that are equivalent by symmetries produce the same result in the simulator. Thus, when the genetic

algorithm evaluates a mapping, if it has already evaluated another mapping which is equivalent to it, it can retrieve the result from the symmetry-aware cache. Since only the cache is aware of the symmetries, the genetic algorithm behaves identically with this symmetry-aware cache and with a regular one: it produces exactly the same results.

For each architecture topology, we do a DSE for each application of the E3S benchmark suite, once using a regular and once a symmetry-aware cache. We run the genetic algorithm for 10 generations with 10 offspring each. To account for the randomness in the algorithms, we execute each algorithm with 10 different random seeds. Figure 10 shows the effect of a symmetry-aware cache for pruning the design-space exploration for the different architectures. We normalize the execution times for each execution of the DSE algorithm (with a concrete random seed, application and architecture), such that the time of the execution with a regular cache is 1. The figure shows the (geometric [23]) mean of the relative execution time of the whole benchmark suite over all runs with the symmetry-aware cache. Note that this refers to the execution time of the DSE itself, not the results of the simulation. The results are identical when using a symmetry-aware or a regular cache. We also disambiguate the execution time of the exploration with a symmetry-aware cache in the actual exploration time, and the overhead from the symmetry calculations. The error bars in the figure show the estimated variance from the multiple random executions and different benchmarks.

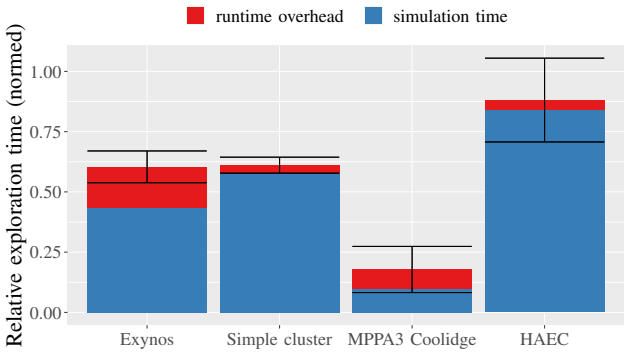

Fig. 10. Exploration time reduction via symmetry-aware caching.

We can see in Figure 10 that using `mpsym` improves the execution time in all cases, especially in the case of complex topology of the Kalray MPPA3 Coolidge. Here, our algorithm improves the execution time by a factor of $5.6\times$. The DSE for the Exynos architecture is also considerably faster with a symmetry-aware cache, reducing the time to slightly less than half, on average. The results are less impressive for the HAEC architecture, which is similarly complex to the Kalray MPPA3 Coolidge, yet using `mpsym` still yields a net improvement. The symmetries of the NoC topology in the HAEC clusters are not completely captured by its automorphism group [12], [10]. The NoC mesh in the HAEC architecture also has partial symmetries that can be leveraged in a non-trivial fashion (cf. Section III). The same is true for the topology of the stacked clusters, where the topmost cluster cannot communicate directly via the wireless inter-board links to the one in the bottom (cf. Figure 1). We expect that using computational methods in inverse semigroups [24] could yield greater improvements this case, but these are not fully implemented in `mpsym` yet. Moreover, the computational overhead for these methods is considerably higher than for the special case of groups. It is less clear that the trade-off would be beneficial for DSE, at least not without algorithmic innovations in the comparatively less-studied computation of partial symmetries.

Using the generic methods of [10], the runtime of the DSE was several orders of magnitude higher than the regular cache for the complex architectures (Coolidge, HAEC). As such, we excluded them from the figure for readability and since executing them on the whole benchmark suite for multiple random seeds would require several weeks of computation. As an example, a single evaluation of the genetic algorithm for the MPPA3 Coolidge takes around a minute with the methods presented here, while it might take over an hour with the methods from [10]. This is in part because the implementation in `mpsym` is more optimized. For the smaller architectures, the methods are more comparable (and in fact, identical, for the Exynos architecture without hierarchy). It is worth mentioning that we do not include application symmetries, e.g. considering data-level parallelism for the mapping symmetries. We cannot extract these from the E3S suite, since it does not include the source code for the tasks, nor do we have methods to do this automatically. This omission stands in contrast to the evaluation in [10], where application symmetries were shown to significantly improve the effectiveness of the methods. Including application symmetries is orthogonal to our contribution in this paper and the methods we present here would also benefit from doing so.

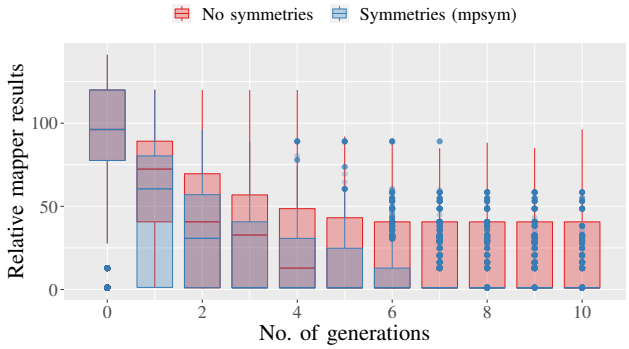

Fig. 11. The effect of symmetry normalization on genetic algorithms on the MPPA3 Coolidge topology as a function of the number of generations.

We can use `mpsym` to improve the results of the algorithm, not only its runtime. For this, we expose only the (lexicographically-minimal) canonical representatives to the algorithms as a mapping representation [15]. In this way we can explore the set of orbits $M \setminus G$ as described in Section IV-B, instead of exploring the whole set M. With the same basic setup as in Figure 10 we now compare the regular variant (without symmetries) with a variant executing the exploration on a pruned design space.

In contrast to the symmetry-aware cache, this does change the results of the DSE. As such, we also consider the simulated

runtime of the best mapping found, on the target architecture, which quantifies the quality of the result. In contrast, we also compared the exploration time for the DSE on the host machine. Note that these two values are fundamentally not comparable, since they refer to different times. One is simulated and pertains the benchmark, while the other one is measured and refers to the DSE itself. Nevertheless, lowering both times is relevant to improving DSE.

Since the times are different for the different applications, we again consider relative times, showing the relative improvement on the quality of the result by using `mpsym`. The results are shown in Figure 11, which reports the relative improvement of the results. For each generation, we show a box-and-whiskers plot with the partial results of the meta-heuristic at that point. The figure only shows the results for the MPPA3 Coolidge. For all other three achitectectures, there was no (statistically significant) difference between the quality of the results. Overall, the results show clearly that larger design spaces benefit more from our method. On average, using `mpsym` improved the quality of the results in the Kalray MPPA3 Coolidge topology by 24.7%.

2) Extending to other Algorithms: A central property of our methods is that they are, to an extent, algorithm-agnostic. In principle, they can be used to improve any meta-heuristic that has a regular structure in its search. To show this, we implemented two additional mapping algorithms: a simulated annealing algorithm based on [25] and tabu-search, following the method proposed in [26]. We also implemented a simple random walk search as a baseline. Both setups evaluated in the previous section can be readily used for other algorithms. We start by evaluating if `mpsym` can accelerate these other algorithms by using a symmetry-aware cache. Other than the symmetry-aware cache, the algorithms are completely unmodified and produce the exact same results when using `mpsym`.

Figure 12 shows the results of a symmetry-aware cache to the other algorithms. This figure is analogous to Figure 10, extended to the other algorithms. For the random walk heuristic, adding a symmetry-aware cache with `mpsym` did not change the simulation time. With the additional overhead from the symmetries computations, this algorithm performed worse by using `mpsym`. This is to be expected, since a random walk is a very unstructured search. Algorithms like tabu search or simulated annealing, on the other hand, work on the basis of a local search and will tend to compare similar mappings in the process. In this setting, a symmetry-aware cache is more profitable, as can be seen in the figure. Again, the results are best in the complex and hierarchical topology of the Kalray MPPA3 Coolidge architecture. Concretely, the execution time of the simulated annealing heuristic was improved by a factor of $8.55\times$.

When exploring the HAEC topology, on the other hand, the marginal improvements in execution time are not enough to offset the overhead from `mpsym`'s symmetry calculations. As explained above, we believe this to be a consequence of the missing partial symmetries in the current implementation. For the smaller architectures (Exynos, multi cluster) we see that changing operations made a modest difference only for the

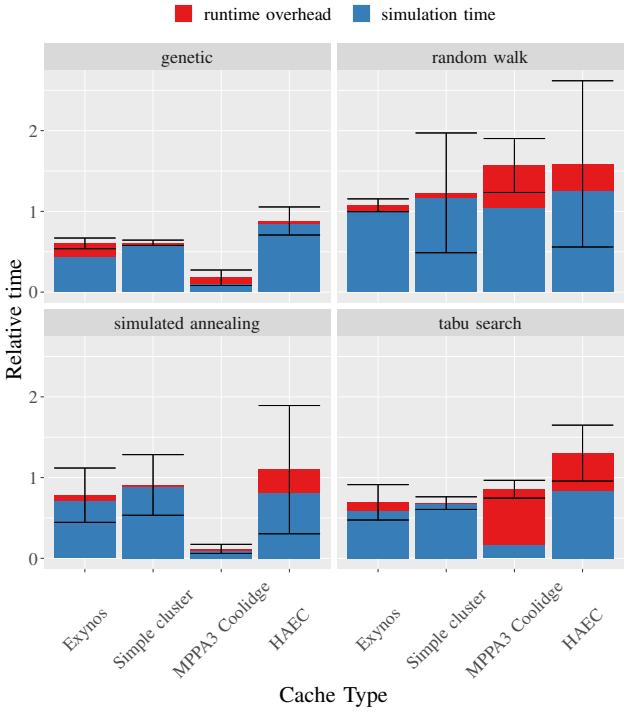

Fig. 12. The effect of caching on multiple algorithms.

algorithms. Most importantly, the experiment shows how our method indeed works for multiple algorithms, benefiting three completely different meta-heuristics without changing them at all.

In some cases, like tabu search on the MPPA 3 Coolidge, the cache overhead surpasses the simulation time. The figure shows, however, that despite the overhead, the method yields an overall improvement, reducing the total exploration time.

We also extend the evaluation of the improved exploration in the factor space to the other mapping meta-heuristics. We cannot directly extend Figure 11 to the other meta-heuristics. The iterations of the different algorithms like generations, or iterations of the tabu search and simulated annealing, are fundamentally different and not comparable. Moreover, the number of mappings evaluated by the heuristics is dynamic and depends on the concrete values of the exploration. As such, we cannot set the numbers of explorations to be identical for the different heuristics. Nevertheless, we set the parameters to attempt to have a similar number of simulations per meta-heuristic. We set the number of random walk iterations to 300. For the tabu search algorithm, we set the maximum number of iterations to 5, each of size 5, with a tabu tenure of 5 and a move set of size 10. The simulated annealing heuristic was the one with the least predictable iteration numbers. To attenuate this, we set the initial temperature equal to the final temperature as 0.1, and a temperature proportionality constant of 0.5.

Figure 13 gives an overview of the improvements of using `mpsym` in the different algorithms. This includes the results in terms of the goodness of mappings. It also includes the changes in relative exploration time, for reference between the algorithms. The values are all normed such that the genetic

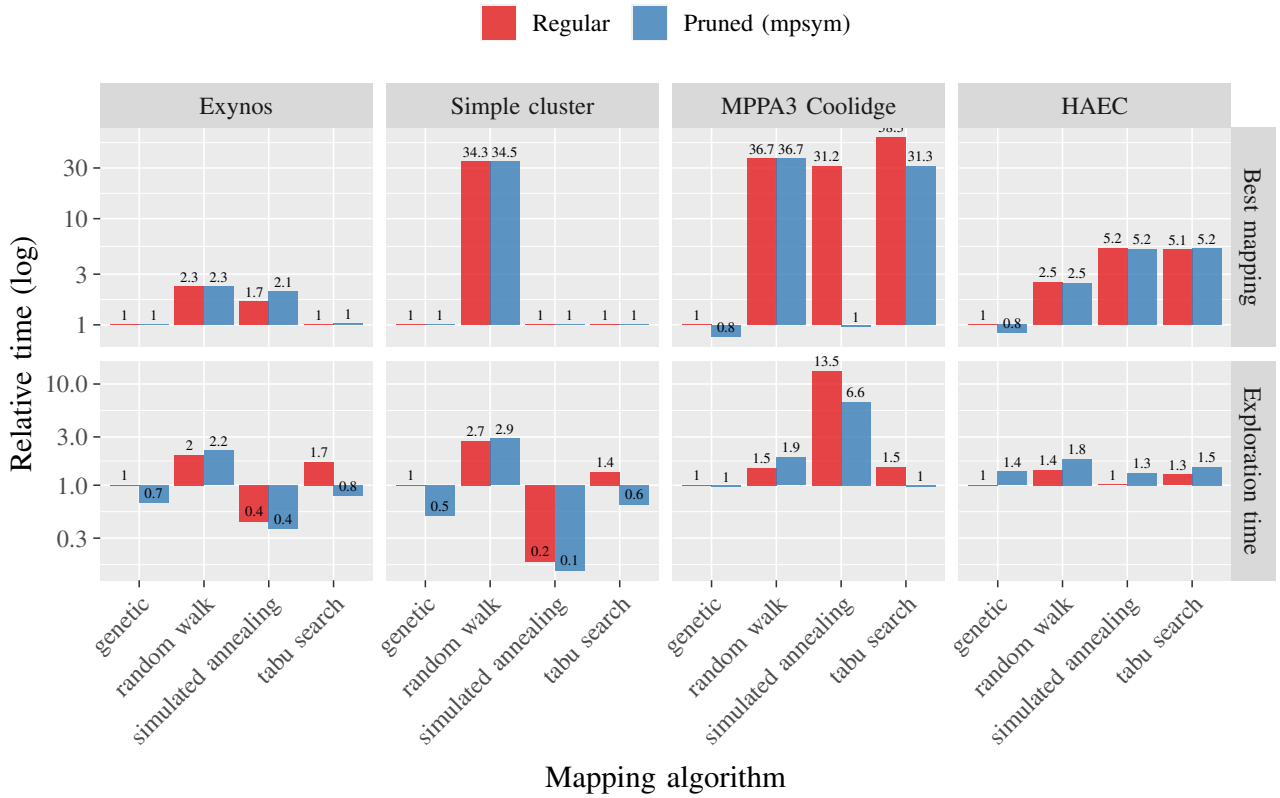

Fig. 13. Overall improvements of symmetry normalization on multiple algorithms.

algorithm without the symmetry pruning has a (relative) time of 1, both for the simulated time of the best mapping and for the exploration time of the DSE. This also allows us to compare between the different mapping algorithms. The (relative) exploration times reported for the standard algorithms include the symmetry-aware cache and are thus still mostly faster than the completely unmodified algorithms.

The overall improvement of using `mpsym` in terms of the quality of the results is modest for the simpler topologies (Exynos, simple cluster), yet consistent. Even for these architectures, the methods are useful. However, the results are most impressive for the Kalray MPPA3 Coolidge architecture, where the simulated annealing heuristic found solutions which were, in average, 32.4× better than without `mpsym`. In other words, in less time as the unmodified version and without changing the algorithm, our methods improved this algorithm by an average of 32.4 ×. The mapping meta-heuristics considered perform extremely poorly on the enormous design-space that arises from complex topologies like that of the MPPA3 Coolidge architecture. In that light, the strength of the results is perhaps primarily an indication of the problem posed by complex topologies in design-space exploration.

D. Scaling

The focus of this paper is the complexity in the architectures, more so than the applications. Nevertheless, discussing how the size of the application affects the usefulness of the methods is still a relevant question, albeit not the central one.

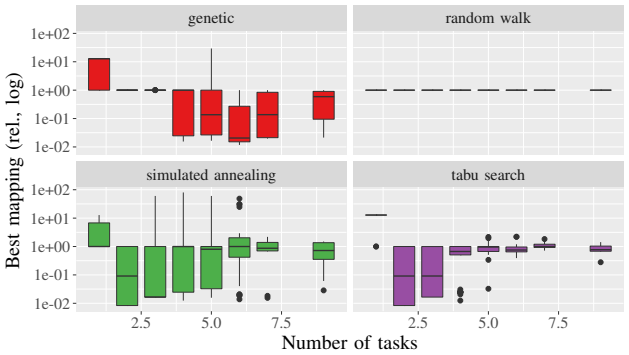

Fig. 14. The effect of symmetry normalization on multiple algorithms on the MPPA3 Coolidge topology.

Figure 14 shows the relative results of the DSE, in terms of the simulated time achieved by the best mapping found, as a function of the number of tasks in the benchmark. We see that the improvements are less pronounced for larger numbers of tasks. However, the largest benchmark in the suite has 9 tasks, which is less than the number of cores in a single cluster of the MPPA3 Coolidge (17). Thus, the E3S suite, while useful for having multiple benchmarks from different domains, is not ideal for evaluating the limits of our method.

To evaluate how our approach scales with larger applications, we used the open-source tool SDF^3 [27] to generate random Synchronous Data Flow (SDF) graphs. These graphs can also be read and simulated by mocasin [21]. We generated

graphs of increasing sizes, $2, 4, 8, 16, 32, 64$ and 128 actors[6]. We executed a DSE flow for each of these sizes on the Kalray MPPA3 Coolidge model to evaluate the scalability of the method for larger task graphs.

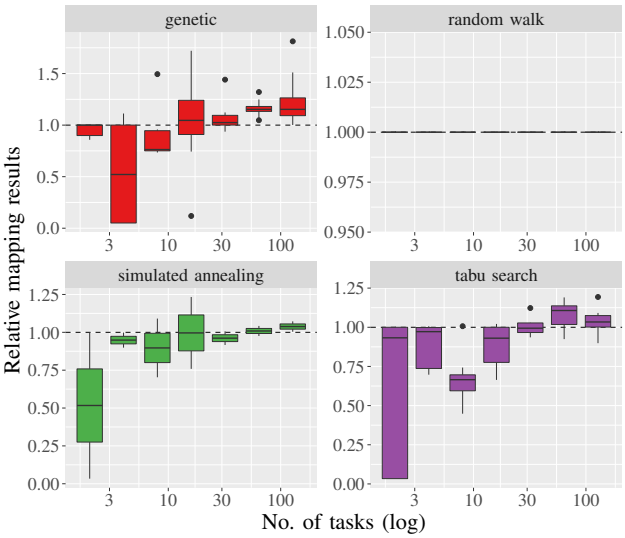

Fig. 15. Scaling of symmetries to the relative mapper performance of random SDF graphs on the MPPA3 Coolidge Platform.

Figure 15 shows the relative results of the DSE for the random SDF graphs, in terms of the best mapping found (simulated time). A dotted line at 1 shows the point where the algorithms on the pruned space are equal to the regular algorithms without symmetries. We can see that for moderate amounts of tasks, between 16 and 32, using `mpsym` results in an improvement. The method stops improving, and in fact at some points being detrimental, for very large graphs ($64-128$ actors). It is safe to assume that for these very large graphs both variants perform poorly, since the unmodified variant already does so for low numbers of actors. An important point to note here is that it is very likely that applications with hundreds of tasks will probably have some data-level parallelism, for which we could use application symmetries as described and demonstrated to be useful in [10].

Figure 16 shows the effect of the number of tasks on the other metric of DSE improvement, the relative exploration time on the host machine. For the genetic algorithm, the symmetries continue to be useful even for very large graphs, while for the other algorithms there is at least no detrimental effect. These results indicate that the methods scale better with the architecture complexity than the application size. As mentioned above, we believe including application symmetries can make the methods scale also well with increasing application complexity.

When considering the scalability from the architecture's perspective, specifying concrete time complexities is difficult, particularly for local search. This is the case because there is a runtime/accuracy trade-off involved. For iteration, which seems to be better than orbit enumeration, space complexity is basically constant, since the size of the BSGS size does not

[6]The tasks in the SDF model, as in many dataflow models, are usually called actors

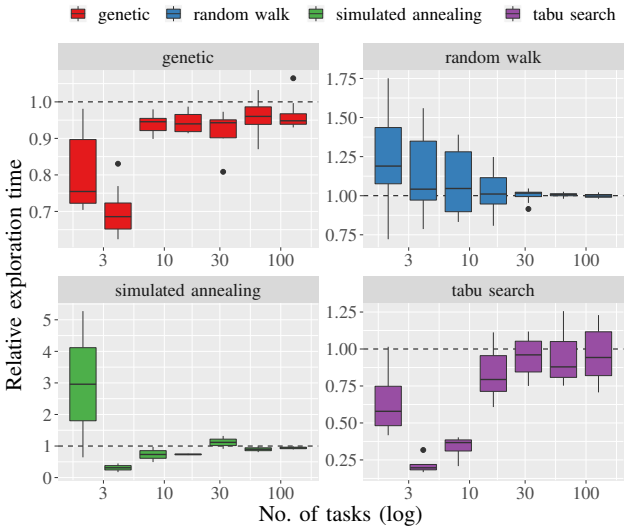

Fig. 16. Scaling of symmetries to the relative exploration time of random SDF graphs on the MPPA3 Coolidge Platform.

really scale with group order for the groups involved. In this case, we only need to store a BSGS for each proto/super graph and generators are created from these. Time complexity is a function of $O(|H|)$ and $O(|G|)$, for $G \wr H$. More importantly though, the actual complexity as a function of the number of PEs depends on the symmetries themselves. Nevertheless, the results of this evaluation seem to indicate that the approach does scale in practice to large and complex hierarchical architectures.

VI. Conclusion and Outlook

In this paper we have presented `mpsym`, a C++ library implementing novel domain-specific algorithms for accelerating and improving DSE of complex, hierarchical architectures. We have seen how our design integrates with existing algorithms without modification, improving them both in terms of runtime and the quality of the results. An evaluation on the E3S benchmark significantly improved the DSE in the clustered topology of the Kalray MPPA3 Coolidge, by an order of magnitude in terms of exploration time, and three orders of magnitude in terms of the quality of results. This is probably mostly a testament of how regular algorithms struggle to deal with complex topologies with multiple levels of hierarchy.

The methods presented here are well-suited to deal with multiple levels of hierarchy by using a wreath-product decomposition of the symmetry group. However, they still do not achieve maximal performance on topologies with non-trivial partial symmetries, like regular mesh NoC topologies. In future work, we plan to use partial symmetries and the theory of inverse semigroups to improve these cases as well.

The evaluation showed how the methods scale well with increasing architecture complexity, yet they struggle with increasing application sizes. We believe including application symmetries should mitigate this, helping with scalability also for complex applications.

ACKNOWLEDGMENT

This work was funded in part by the German Research Council (DFG) through the TraceSymm project (number 366764507) and the Studienstiftung des deutschen Volkes.

REFERENCES

[1] Kalray. (2020) Mppa3 coolidge. [Online]. Available: https://www.kalrayinc.com/download/mppa-processor-flyer/

[2] G. Fettweis, M. Dörpinghaus, J. Castrillon, A. Kumar, C. Baier, K. Bock, F. Ellinger, A. Fery, F. H. P. Fitzek, H. Härtig, K. Jamshidi, T. Kissinger, W. Lehner, M. Mertig, W. E. Nagel, G. T. Nguyen, D. Plettemeier, M. Schröter, and T. Strufe, "Architecture and advanced electronics pathways towards highly adaptive energy-efficient computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 204–231, Jan. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8565890

[3] 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*, 2016, pp. 153–162.

[4] A. Goens, R. Khasanov, M. Hähnel, T. Smejkal, H. Härtig, and J. Castrillon, "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)*, ser. SCOPES '17. New York, NY, USA: ACM, Jun. 2017, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/3078659.3078663

[5] A. K. Singh, A. Kumar, and T. Srikanthan, "Accelerating throughput-aware runtime mapping for heterogeneous mpsocs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 1, pp. 1–29, 2013.

[6] M. Thompson and A. D. Pimentel, "Exploiting domain knowledge in system-level mpsoc design space exploration," *Journal of Systems Architecture*, vol. 59, no. 7, pp. 351–360, 2013.

[7] R. Piscitelli, "An examination of keystroke dynamics for continuous user authentication," Ph.D. dissertation, University of Amsterdam, 2014. [Online]. Available: https://hdl.handle.net/11245/1.430852

[8] T. Schwarzer, A. Weichslgartner, M. Glaß, S. Wildermann, P. Brand, and J. Teich, "Symmetry-eliminating design space exploration for hybrid application mapping on many-core architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 297–310, 2017.

[9] V. Richthammer, T. Schwarzer, S. Wildermann, J. Teich, and M. Glaß, "Architecture decomposition in system synthesis of heterogeneous many-core systems," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[10] A. Goens, S. Siccha, and J. Castrillon, "Symmetry in software synthesis," *ACM Transactions on Architecture and Code Optimization (TACO),*, vol. 14, no. 2, pp. 20:1–20:26, Jul. 2017.

[11] V. Richthammer, F. Fassnacht, and M. Glaß, "Search-space decomposition for system-level design space exploration of embedded systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 2, pp. 1–32, 2020.

[12] M. V. Lawson, *Inverse semigroups: the theory of partial symmetries*. World Scientific, 1998.

[13] B. D. McKay and A. Piperno, "Practical graph isomorphism, {II}," *Journal of Symbolic Computation*, vol. 60, no. 0, pp. 94 – 112, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0747717113001193

[14] A. Goens, C. Menard, and J. Castrillon, "On compact mappings for multicore systems," in *Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*, D. Pnevmatikatos, M. Pelcat, and M. Jung, Eds., vol. 11733, IEEE. Springer, Cham, Jul. 2019, pp. 325–335. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-27562-4_23

[15] ——, "On the representation of mappings to multicores," in *Proceedings of the IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-18)*, Vietnam National University, Hanoi, Vietnam, Sep. 2018, pp. 184–191.

[16] Á. Seress, *Permutation group algorithms*. Cambridge University Press, 2003, vol. 152.

[17] *GAP – Groups, Algorithms, and Programming, Version 4.11.0*, The GAP Group, 2020. [Online]. Available: https://www.gap-system.org

[18] A. F. Donaldson and A. Miller, "On the constructive orbit problem," *Ann Math Atrif Intell*, vol. 57, pp. 1–35, 2009.

[19] D. F. Holt, *Handbook of Computational Group Theory*. CRC Press, 2005.

[20] R. Dick. (2008) Embedded systems synthesis benchmark suite (e3s). [Online]. Available: http://ziyang.eecs.umich.edu/~{}dickrp/e3s/

[21] C. Menard, A. Goens, G. Hempel, R. Khasanov, J. Robledo, F. Teweleitt, and J. Castrillon, "Mocasin — rapid prototyping of rapid prototyping tools: A framework for exploring new approaches in mapping software to heterogeneous multi-cores," in *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, 2021, pp. 66–73.

[22] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel, "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, pp. 358–374, 2006.

[23] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: the correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, 1986.

[24] J. East, A. Egri-Nagy, J. D. Mitchell, and Y. Péresse, "Computing finite semigroups," *Journal of Symbolic Computation*, vol. 92, pp. 110–155, 2019.

[25] H. Orsila, T. Kangas, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, "Automated memory-aware application distribution for multi-processor system-on-chips," *Journal of Systems Architecture*, vol. 53, no. 11, pp. 795–815, 2007.

[26] S. Manolache, P. Eles, and Z. Peng, "Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, pp. 1–35, 2008.

[27] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006, pp. 276–278. [Online]. Available: http://www.es.ele.tue.nl/sdf3