# Exploiting Hierarchical Configuration to Improve Run-Time MPSoC Task Assignment*

V. Nollet[1], P. Avasare[1], D. Verkest[1,2], H. Corporaal[3]
[1]IMEC V.Z.W., Kapeldreef 75, 3001 Leuven, Belgium
[2]Professor at Vrije Universiteit Brussel and at Katholieke Universiteit Leuven, Belgium
[3]Professor at Technical University Eindhoven, The Netherlands

## Abstract

*Run-time assignment of a set of communicating tasks onto a heterogeneous multiprocessor system-on-chip (MPSoC) platform is a challenging task. Having FPGA fabric tiles in such MPSoC platform increases performance and flexibility of the platform. Such FPGA tiles can not only run tasks in hardware but also instantiate a soft IP core that executes the task functionality. Thus fully exploiting the available FPGA fabric resources must include exploiting such a hierarchical configuration. This paper details the benefits of using a hierarchical configuration and illustrates how to incorporate it within a generic run-time task assignment heuristic. We show that adding a hierarchical configuration significantly improves task assignment performance (i.e. success rate and assignment quality). In several cases, the performance of a heuristic with a hierarchical configuration extends beyond the capabilities of a full solution space exploration without hierarchical configuration, at only a fraction of the computation time.*

## 1. Introduction

In order to meet the ever-rising compute requirements while retaining platform flexibility, System-on-Chips (SoCs) contain multiple heterogeneous processing elements (PEs or *tiles*). So besides general-purpose processing elements, there will be some specialized processing elements like e.g. DSPs and FPGA fabric tiles [1, 2].

Fine-grain reconfigurable hardware has the ambition to deliver the same amount of flexibility as an Instruction Set Processor (ISP) while providing a performance level close to that of an ASIC. Obviously, these reconfigurable hardware devices operate in a completely different way and, hence, exhibit radically different properties with respect to an ISP architecture.

When combining ISPs with FPGA tiles into a SoC, a platform run-time manager could choose to abstract the architectural differences between the tiles and just consider every tile as a processing element that can execute a task with a certain performance for a certain cost. Hence, one could develop a generic run-time algorithm that optimizes the task assignment according to generic criteria. However, by abstracting the FPGA tile specific properties (e.g. tile size) the task assignment algorithm can be sub-optimal [1].

This paper explores usefulness of exposing the run-time task assignment algorithm to the ability of FPGA tiles to create a configuration hierarchy, i.e. to use FPGA tiles for placing dedicated hardware tasks as well as using them for hosting softcore IP blocks. This opens the door to a novel run-time task assignment algorithm that exploits the hierarchical configuration capabilities. The idea of using hierarchical configuration inside a run-time manager was proposed earlier [1], but was not explored extensively. This paper does that exploration on a large variety of task-sets.

As target platforms we consider a set of heterogeneous multiprocessor systems, each containing various processing element types. The different PE tiles are interconnected by a Network-on-Chip. In this paper, we assume a 3 by 3 mesh network with deterministic XY routing (Figure 1). Nevertheless, the proposed solution is equally applicable on a wide set of platform architectures with different size, topology and routing schemes.

The differences between these processor types will be reflected in (1) their task support, i.e. what percentage of tasks have support for a particular processing element, and (2) their load, meaning that a task will, on average, impose a higher load on a general purpose processor than on e.g. a DSP. Furthermore, we will also make a distinction between a large reconfigurable hardware tile and a small reconfigurable hardware tile. While a large tile can accommodate any FPGA task, the small tile can only accommodate a subset of tasks (i.e. the ones that fit the tile).

Every application that needs to be mapped is described by an acyclic directed graph (further denoted as task graph),

* This work is partly funded by the Flemish Government (GBOU-RESUME project IWT-020174-RESUME).

where each vertex represents an application task and every edge represents a communication link between tasks. A single task can be supported by multiple processing elements.

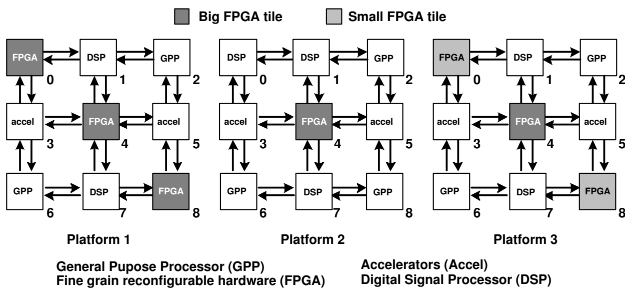


**Figure 1. MPSoC platform architectures, each containing four (hard) PE types.**

The rest of the paper is organized as follows. Section 2 briefly details related work that clearly illustrates the usefulness of exploiting a configuration hierarchy for real-life applications. Section 3 describes the generic task assignment heuristic. Section 4 details the hierarchical configuration rationale and shows how it is integrated with the generic heuristic. Section 5 evaluates the performance of both the generic and the hierarchical heuristic with respect to exploring the full solution space with and without configuration hierarchy. Section 6 presents the conclusions.

## 2. Related Work

Schaumont et al.[3] demonstrate the use of hierarchical configuration on an image processing example application. The author's methodology starts by profiling a set of applications from a target application domain to determine the right point in the configuration design space. This way, a set of commonly used, computationally intensive kernels can be identified. Parameterizable implementations of these kernels form the building blocks of the reconfigurable platform. Consequently, these blocks are pre-instantiated into the FPGA fabric. At run-time these soft IP blocks can be *programmed* with a minimal amount of *configuration* input. The authors suggest to use compiler techniques to map an application onto a given set of parameterizable IP blocks.

The ThumbPod, a real-life embedded fingerprint authentication system, also illustrates the potential of using a configuration hierarchy [4]. Due to the high design complexity, it is next to impossible to capture all functionality in one abstraction level. Instead, the ThumbPod is composed as a stack of three machines: the bottom layer consists of a Virtex-II XC2V1000 FPGA. A LEON2 soft core processor is instantiated on top of the FPGA. On top of the LEON2 processor executes an embedded Java virtual machine. This configuration hierarchy enables easy programming and the use of the the Java security architecture.

Keller et al. [5] describe the use of so-called *software decelerators*. By using freely available soft IP cores, the designer can take advantage of an *easier* application design process (i.e. a software design process instead of a hardware design process). In addition, certain algorithms use less hardware resources when implemented on a soft IP core (i.e. a sequential machine), while still meeting the necessary performance requirements. The authors describe a configuration hierarchy case study using a finite state machine.

Memik et al. [6] describe a system architecture, denoted as *Strategically Programmable System* (SPS), that contains a set of *Versatile Programmable Blocks* (VPB) that are preplaced within the fully reconfigurable logic. In essence, this corresponds to using a configuration hierarchy. When implementing an application, functionality will be mapped onto those VPBs at design-time. This way, not only the amount of configuration bits required to represent an application can be drastically reduced, but also it results in a diminished configuration time (i.e. application setup time).

All the related work mainly considers using a configuration hierarchy from a design-time point of view. Their purpose is to design (a) in a more efficient way by separating complex issues, (b) in a faster way by using a software design process and (c) in a more resource-efficient way. Although task assignment is typically part of a run-time resource manager, none of the authors considered a run-time manager for controlling the configuration hierarchy.

Smit et al. [2] describe a run-time task assignment algorithm designed to map a task-graph at run-time to a tiled heterogeneous platform containing different ISP types as well as FPGA fabric tiles. The MinWeight algorithm takes only a few milliseconds to come up with a mapping solution. The algorithm takes the scarcity of resources into account. This means that, quite similar to our generic heuristic approach, the algorithm assigns a tasks that need *a scarce resource* before all other tasks. Although the authors clearly target architectures containing reconfigurable hardware and although they acknowledge that scarcity of resources can be problematic for the performance of the algorithm, they do not propose to adjust their algorithm with respect to the specific reconfigurable hardware tile properties.

## 3. Generic Task Assignment Heuristic

This section briefly describes our generic task assignment heuristic that can be used to map set of communicating tasks at run-time on an MPSoC platform [1].

At design-time, it is often unknown which applications will run simultaneously. So, only at run-time the resource usage of the platform as well as the application user requirements are known. Hence, one needs a run-time application resource assignment algorithm. A fast, lightweight heuristic that comes up with a reasonably good solution is pre-

ferred over an algorithm that comes up with an optimal solution requiring a lot of computation time.

In order to verify the performance of the proposed heuristic, we also created a *full search* (fs) algorithm. This algorithm exhaustively searches all possible solutions for assigning an application task graph to a multiprocessor platform with a certain load state. By using the full search algorithm one can verify if an assignment is at all possible in case the heuristic does not find a suitable assignment. The full search algorithm determines the quality of a total task graph assignment based on the product of the processor load variance, the communication load variance and the hop-bandwidth product (i.e. the product of the communication load between two tasks and the hop-distance between them). Furthermore, it allows to assess the quality of the assignment solution provided by the heuristic. To this end the full search algorithm records the maximum and minimum hop-bandwidth over all feasible task assignment solutions for a single task graph.

For assigning platform resources to a new incoming application, the heuristic requires a specification of the application, a description of the platform, the user requirements with respect to the application and, finally, the current resource usage of the platform. The heuristic uses the following steps to assign a task graph (white blocks in Figure 4).

1. **Calculate task priority.** This is done in two steps. (1) For every task $T_i$ in the application, determine its load variance with respect to the different supported PE types and normalize that value by the number of evaluated PE types ($V_{Ni}$). Tasks with a high $V_{Ni}$ are very sensitive to which processing element they are assigned to. In addition, tasks that can only be mapped on one specific PE should be mapped before all other tasks. This way, the heuristic avoids a mapping failure, that would occur if this specific PE would be occupied by another task. (2) For every task $T_i$ in the application, determine its communication importance $C_i$ (both incoming and outgoing) with respect to the total inter-task communication of the application. This allows the algorithm to order the tasks based on their communication requirements.

2. **Sort tasks according to mapping importance.** The assignment priority of a task $T_i$ is equal to $V_{Ni}$ x $C_i$. Tasks are sorted by descending priority.

3. **Sort PEs for most important unmapped task.** This step contains two phases. First, the goodness of the PEs for a task $T_i$ is determined based on the product of the current PE load and the already used communication resources to its the neighboring tiles. Secondly, in order to map heavily communicating tasks close together, the goodness is also multiplied with the hop-bandwidth product to its already assigned communication peers. Solutions that lack the required computation (phase 1) or communication (phase 2) resources have their goodness set to zero, indicating that the PE is not fit to accommodate the unmapped task.

4. **Mapping the task to the best computing resource.** The most important unmapped task is assigned to the best fitting PE. Consequently, the platform resource usage (i.e. PE load and load of the communication links) is updated to reflect this assignment. Steps 3 and 4 are repeated until all tasks are mapped.

Occasionally this greedy heuristic is unable to find a suitable assignment for a certain task. This usually occurs when mapping a resource-hungry application on an already heavily loaded platform. *Backtracking* is the classic solution for this issue: it changes one or more previous task assignments in order to solve the mapping problem of the current task.

The backtracking algorithm starts by finding a previously assigned task $T_i$ with multiple assignment options (i.e. more than one suitable PE for assignment). Consequently, all resource allocations up until this task are undone. Then, task $T_i$ is assigned to the *second best* PE. From then on, the heuristic starts all over for task $T_{i+1}$. Backtracking stops when either the number of allowed backtracking steps is exhausted or when backtracking reached the first task assignment of the application.

When that happens, the algorithm can (a) use run-time task migration [1] to move a task of a previously mapped application or (b) restart the heuristic with reduced user requirements. However, the assignment success rate and the quality of the solution of the heuristic can be significantly improved by using hierarchical configuration.

## 4. Hierarchical Configuration

This section first details the rationale of using a configuration hierarchy and then explains how to incorporate it within the generic task assignment algorithm.

### 4.1. Rationale

Historically, FPGA fabric allowed to separate the design from the actual physical hardware. In recent years, FPGAs have become large and fast enough to accommodate programmable IP cores. Hence, the boundary between software, executing on the IP core, and *soft hardware*, instantiated in the FPGA is fading.

In order to understand the ability of FPGA to create a configuration hierarchy, consider an FPGA that runs a programmable soft IP core (Figure 2). In turn this soft IP core executes some user program. While the soft IP core acts as *program code* for the FPGA fabric, it also acts as hardware

for the user program. Consequently, the user program defines the actions for the programmable IP core, while from an FPGA point of view it is merely data being processed by the soft IP core circuit. This setup forms a *configuration hierarchy*.

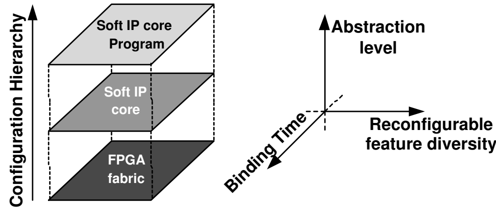


**Figure 2. Configuration hierarchy concept and design space description.**

Schaumont et al. [3] coined the term *hierarchical configuration* and described the configuration design space by means of three axis (Figure 2). The *vertical* axis describes the level of abstraction. From a software point of view, it corresponds to having a virtual machine executing instructions using the functionality and primitives provided by the underlying abstraction layer. The *horizontal* axis describes the reconfigurable feature diversity. This axis is typically associated with terms as coarse-grained and fine-grained reconfigurability with respect to communication, computation and storage elements. The *time* axis denotes the binding time, i.e. the time when configuration data is sent to the processing part.

In recent years, most FPGA vendors provide a wide range of soft IP components for e.g. data encryption and signal processing. In addition, there is a growing community of open source softcore IP blocks. This is an enabling factor for building a softcore IP library to be used by the run-time manager.

Using such a soft IP core (also denoted as a *software decelerator* [5]) often results in a speed/performance penalty with respect to instantiating a hardware circuit with the same functionality.

From a design-time point of view, however, there is a trade-off between performance and other costs such as chip area needs or ease and speed of implementation [5, 7]. Meaning that creating a software task is far easier than creating a dedicated hardware task.

From a run-time point of view, using a soft IP core can result in more efficient usage of the platform FPGA resources. First, it enables time-multiplexing, i.e. having multiple tasks using FPGA resources in a concurrent way. Secondly, it greatly improves spatial freedom for task assignment, meaning that tasks can be placed more freely and communication resource bottlenecks can be circumvented.

Hence, the ability to use a configuration hierarchy creates significant design-time and run-time opportunities. However, these opportunities require a run-time manager capable of controlling such hierarchy.

## 4.2. Improving the Generic Heuristic

Before deciding on *how* to add hierarchical configuration support to the heuristic, one needs to determine if it make sense and, if so, in what occasions it is useful. To answer these questions, we added hierarchical configuration to the full search algorithm and performed a set of experiments using the experimental setup detailed in Section 5.1.

These experiments (using platform 1 of Figure 1 and a high application load) revealed that between 57% and 75% of the solutions selected as best by the full search algorithm used a configuration hierarchy. Consequently, we analyzed the properties of the tasks that were assigned to a softcore and we performed a qualitative analysis of the task graphs assigned to a configuration hierarchy.

We noticed that on average 86% of all tasks that were assigned to a softcore, supported only one non-FPGA hard PE type (i.e. GPP, DSP or accelerator).

When the FPGA task does not fit on a small FPGA tile, it might be possible to assign this task to a softcore that, in turn, *does* fit the small tile. When analyzing the effects of hierarchical configuration using platform 3 (i.e. containing two small FPGA tiles), we noticed that such situations do occur.

The qualitative analysis of the assigned task graphs showed that hierarchical configuration is mainly used in case of on-chip communication constraints.

First, this occurs when a task has a lot of communication peers, but does not have reconfigurable hardware support. Ideally, this task is assigned to tile 4 (i.e. the center tile). This becomes a possibility using hierarchical configuration. This assignment should, on average, also reduce the hop-bandwidth product.

Secondly, using hierarchical configuration occurs when an assignment fails because of a communication load constraint. Consider the (simplified) assignment example (Figure 3a), where task $T_C$ still needs to be assigned. Assuming $T_C$ has no FPGA support (tile 4) and all other tiles are occupied or unsupported, then $T_C$ can only be assigned to either tile 5 or tile 7. Although both tiles can provide the required computing power, there is a problem with respect to platform communication resources to support the communication between $T_B$ and $T_C$. Without hierarchical configuration, the heuristic has no other option but to reconsider the assignment of $T_A$ and $T_B$ (i.e. perform backtracking) or to migrate tasks of previously assigned applications. However, by means of hierarchical configuration, $T_C$ can be mapped on a softcore instantiated on FPGA tile 4 (Figure 3b). Also

from a hop-bandwidth point of view (i.e. assignment quality), it is better to map $T_C$ on a softcore on FPGA tile 4 than on tile 5 or tile 7.

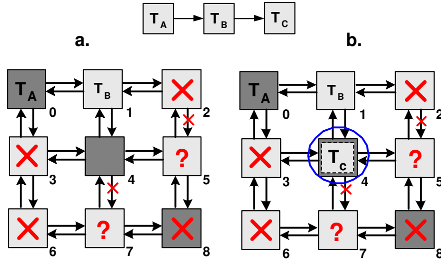


**Figure 3. Hierarchical configuration example.**

Now that we determined the benefits of using a configuration hierarchy, we still need to decide on how to introduce it into the generic heuristic. The initial idea of handling it as an alternative to backtracking [1] (i.e. using hierarchical configuration when backtracking fails), does not yield the expected performance increase. Consequently, we added hierarchical configuration into the main flow of the heuristic as shown by gray blocks in Figure 4. *Before* step 3 of the generic heuristic, we now perform the following steps.

1. **Sort the softcores for most important unmapped task** In case the task only supports one hard PE, we determine a priority on its supported softcores. This entails taking the softcore size into account with respect to size variance of the platform FPGA tiles. In case the softcore supports multitasking, one also needs to consider the re-usability of the softcore. This effectively means considering the load that the current task already imposes (i.e. how much can be re-used by another task). In addition, one needs to take the overall support factor for that softcore into account in order to increase the possibility of another task being able to use this softcore. Obviously, all softcores that cannot deliver the required task performance are neglected. Finally, we end up with a sorted list of softcores.

2. **Instantiate softcores** In this step we instantiate a softcore on every available reconfigurable hardware tile if the task does not have FPGA tile support. The softcores are chosen based on the priority list. Before instantiating the softcore, we need to make sure that (1) the softcore fits on the tile and (2) the softcore will, once instantiated, provide the required performance for the task. This evaluation is especially needed when softcores have a different performance depending on the host FPGA tile. In case the task does have FPGA tile support, a softcore is only instantiated on the tiles that are too small to fit the FPGA task (i.e. the FPGA tiles that would otherwise be unusable).

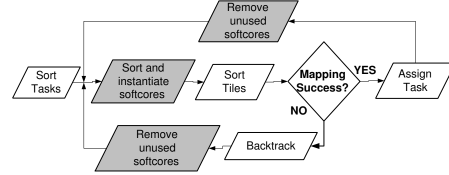


**Figure 4. Generic heuristic (white) with extra steps for hierarchical configuration (gray)**

After these two steps, we perform step 3 and step 4 of the generic heuristic. Meaning that we use the regular algorithm to determine a task assignment. These steps just treat the instantiated softcores as if they were regular hard ISPs.

After the task is assigned (step 4) we still need an extra step, namely to remove all unused softcores in order to expose the FPGA fabric again.

This approach also allows to combine backtracking and hierarchical configuration. If, during backtracking, a task is removed from a softcore and if no other task still uses that softcore, the softcore itself is also removed. This way, the heuristic can opt for a solution without a softcore.

## 5. Performance Evaluation

This section details the experimental setup and compares task assignment performance of the algorithms with and without hierarchical configuration.

### 5.1. Experimental Setup

To evaluate the task assignment performance, a large set of task graphs with various properties is required. We used a software tool called *Task Graphs For Free* (TGFF) [8] to create 1000 random task graphs, each containing between 3 and 10 tasks and where every task contains up to three communication links. Every task potentially has multiple implementations in order to support up to four processing element types (hard or soft), which includes minimal one hard PE type (i.e. GPP, DSP, FPGA or Accel, see Figure 1). As noted previously, we assume a 3 by 3 mesh network with deterministic XY routing (Figure 1).

Depending on the user requirements (high or low), every task implementation as well as every communication link is assigned a certain random load. Table 1 details the average load a task imposes on a processing element type with respect to the user requirements. In addition, Table 1 also details the task support rate, indicating that e.g. 6 out of 10 tasks have support for executing on the GPP. Our softcore library contains seven softcore PEs of various sizes. A random load value is also assigned to every application com-

munication link: on average 25% and 50% load for respectively low and high user requirements.

| PE Type | Low load | High load | Task support |
|---|---|---|---|
| GPP | 25% | 50% | 60% |
| DSP | 20% | 45% | 40% |
| FPGA | 18% | 30% | 20% |
| Accel | 15% | 30% | 20% |
| SoftPE$_i$ | 37% | 80% | 5%-15% |

**Table 1. Task properties w.r.t the PEs**

The platform resources are pre-loaded to indicate the presence of previously assigned applications. The *low* and the *high* platform load parameter indicate that no platform resource (both computation and communication) is used for more than respectively 25% and 50%. A random function determines the actual resource usage for every resource. Due to the binary load state of FPGA tiles (i.e. either 0% or 100% load), they are always set as free. This way, we can clearly determine the effect of changing the amount of FPGA tiles or their size.

## 5.2. Generic Heuristic

Figure 5 details the assignment success rate of the generic heuristic for platform 1. The success rate is a function of the new application load (first letter) and the current platform PE and communication link load (second and third letter). Platform 2 and 3 (Figure 1) yield similar results. The results are relative to the performance of the full search algorithm, i.e. 100% denotes all situations where a full search algorithm could find at least one suitable assignment. The amount of allowed backtracking steps is indicated between brackets.

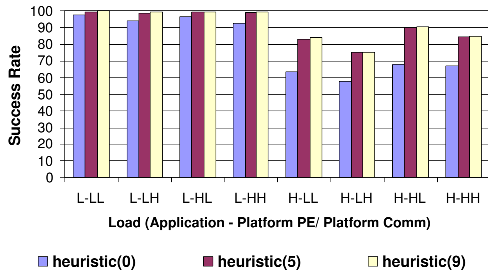


**Figure 5. Generic heuristic success rate on platform 1 relative to full search (100%).**

It shows that the generic heuristic nearly always finds a solution for low application load irrespective of the platform load. In case of high application load the heuristic does not perform as good as the full search (fs) algorithm, but it is successful in at least 75% of the experiments. Hence, the hierarchical configuration experiments of Section 5.3 focus on the case of high application load.

The speed of the heuristic lies well within acceptable run-time boundaries when running on a typical embedded processor like the StrongARM (206 MHz). The heuristic requires on average (depending on the platform load) between 141 $\mu$s and 169 $\mu$s (stdev about 100 $\mu$s) to reach an assignment using at most 9 backtracking steps. In contrast, the full search algorithm requires on average between 2.7 ms and 15.9 ms with peaks reaching up to 4 seconds when faced with a low platform load and a large amount of assignment options.

Besides success rate and calculation speed, the quality of the results produced by the algorithm is equally important. Figure 6 details the hop-bandwidth product (average over all experiments). We do not only compare the result of the heuristic with the best result chosen by the full search (fs) algorithm, but also provide the maximum and minimum of the hop-bandwidth obtained during the exploration of the full solution space. This puts the quality of the chosen solution in perspective with all potential solutions.

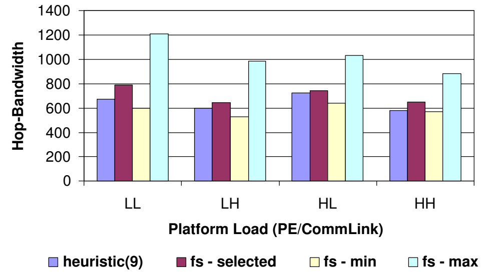


**Figure 6. Hop-bandwidth quality (platform 1).**

We can conclude that the quality of the results produced by the heuristic are quite close to the those of the full search algorithm. On average, the solutions selected by the heuristic yield a lower hop-bandwidth product with a higher communication load variance.

## 5.3. Heuristic with Hierarchical Configuration

The following results detail the (absolute) success rate for both the heuristic and the full search algorithm, both without and with configuration hierarchy support (denoted with *-ch*) for platform 1, 2 and 3.

For platform 1 we first notice that using a configuration hierarchy clearly improves the task assignment success rate for both the full search (fs) algorithm and the heuristic (Figure 7). Secondly, we see that the heuristic *with* hierarchical configuration support performs better than the full search algorithm *without* configuration support in three occasions. This means that, no matter how one would improve

the generic heuristic, it could never outperform the success rate of the heuristic with hierarchical configuration support.

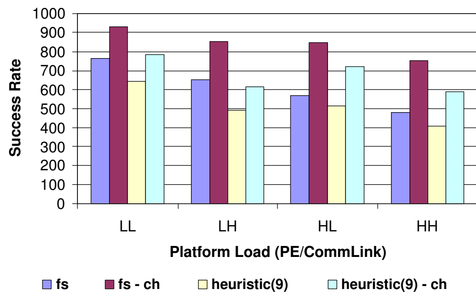


**Figure 7. Hierarchical configuration success rate, high load application on platform 1.**

Platform 2 only contains one reconfigurable hardware tile. Obviously this limits the potential of using hierarchical configuration. Although we notice a significant performance improvement for both the full search algorithm and the heuristic, the heuristic *with* hierarchical configuration never outperforms the full search algorithm *without* configuration hierarchy.

Platform 3 contains, just like platform 1, three FPGA fabric tiles, but two of them are small. This means that several FPGA tasks as well as three out of seven softcores do not fit on these small FPGA tiles. Nevertheless, we notice that the success rate for platform 3 is almost as good as for platform 1. This is due to the fact that if the FPGA tile is not big enough to accommodate the (FPGA) task, in some cases (for about 16 to 22 task graphs) this is solved by using a softcore that does fit the small FPGA tile.

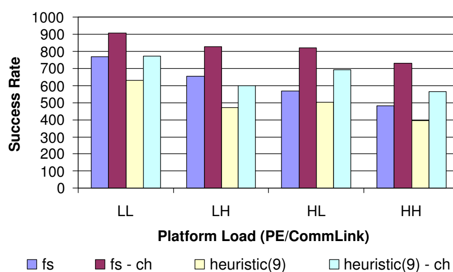


**Figure 8. Hierarchical configuration success rate, high load application on platform 3.**

As Section 4.2 predicts, using a configuration hierarchy reduces the overall average hop-bandwidth product because for some task graphs, tasks can be mapped closer together. For platform 1 the hop-bandwidth reduces up to 8%, while for platform 2, the reduction reaches about 10%. Hence, the hierarchical configuration heuristic combines a higher success rate with a lower average hop-bandwidth. The reduction is higher for platform 2 than for platform 1 because the heuristic spreads the communication load on platform 1 (i.e. a higher use of the corner FPGA tiles). Note that the platform 2 success rate is lower than platform 1.

With respect to speed, to run the hierarchical configuration heuristic on StrongARM (206 MHz) requires on average (depending on the platform type, load and task graph properties) between 179 $\mu$s and 242 $\mu$s (stdev about 180 $\mu$s). On the other hand, exploring the full hierarchical search space can take up to several minutes.

## 6. Conclusion

This paper details the rationale of hierarchical configuration and shows how it can be integrated into a (generic) run-time task assignment heuristic. We show that exploiting a configuration hierarchy, can significantly improve the performance of the run-time task assignment algorithm. This entails increasing its assignment success rate (up to 27% for full search and up to 20% for the heuristic) and improving the task assignment quality (up to 10% depending on the platform). In some cases, the average heuristic success rate improvements even exceed searching the full solution space without hierarchical configuration, while using only a fraction of the execution time. In future, we plan to introduce this algorithm on different real-life MPSoC platforms to evaluate run-time task assignment.

## References


[1] V. Nollet, T. Marescaux, P. Avasare, and J-Y. Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 234–239, 2005.

[2] L. Smit, G. Smit, J. Hurink, H. Broersma, D. Paulusma, and P. Wolkotte. Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture. In *Proc. of Field-Programmable Technology*, pages 421–4, 2004.

[3] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. Quick safari through the reconfiguration jungle. In *Design Automation Conference (DAC)*, pages 172–177, 2001.

[4] P. Schaumont and I. Verbauwhede. Thumbpod puts security under your thumb. *Xilinx Xcell Journal*, 2003.

[5] Eric Keller, Gordon J. Brebner, and Philip James-Roxby. Software decelerators. In *Proc. of Field Programmable Logic and Applications (FPL)*, pages 385–395, 2003.

[6] S. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. Sps: A strategically programmable system. In *Proc. of Reconfigurable Architecture Workshop (RAW)*, 2001.

[7] V. Nollet, J-Y. Mignolet, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Hierarchical run-time reconfiguration managed by an operating system for reconfigurable systems. In *Proc. of ERSA*, pages 81–87, June 2003.

[8] R. Dick, D. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign (CODES)*, pages 97–101, 1998.