Design-time application mapping and platform exploration for MP-SoC customised run-time management

Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor and H. Corporaal

Abstract: In an Multi-Processor system-on-Chip (MP-SoC) environment, a customized run-time management layer should be incorporated on top of the basic Operating System services to alleviate the run-time decision-making and to globally optimise costs (e.g. energy consumption) across all active applications, according to application constraints (e.g. performance, user requirements) and available platform resources. To that end, to avoid conservative worst-case assumptions, while also eliminating large run-time overheads on the state-of-the-art RTOS kernels, a Pareto-based approach is proposed combining a design-time application and platform exploration with a low-complexity run-time manager. The design-time exploration phase of this approach is the main contribution of this work. It is also substantiated with two real-life applications (image processing and video codec multimedia). These are simulated on MP-SoC platform simulator and used to illustrate the optimal trade-offs offered by the design-time exploration to the run-time manager.

1 Introduction
An Operating System (OS, also called run-time management layer) is a middleware acting as a glue layer between both application and platform layers. Just like ordinary glue, an ideal OS should be adapted to the properties and requirements of the environment it has to be used for. In a Multi-Processor System-on-Chip (MP-SoC) environment, this OS should efficiently combine different aspects already present in different disciplines: to implement dynamic sets of applications as in the workstation environment, to manage different types of platform resources as in the parallel and distributed environment and to handle non-functional aspects as in the embedded environment:

• First, mobile systems are typically battery-powered and have to support a wide range and dynamic set of multimedia applications (e.g. video messaging, web browsing, video conferencing), three-dimensional games and many other compute-intensive tasks [1]. These applications are becoming more heterogeneous, dynamic with multiple use cases and data-intensive. Hence, MP-SoC platforms have to be flexible and to fulfill Quality-of-Service (QoS) requirements of the user (e.g. reliability, performance, energy consumption and video quality). Also the OS must be able to run all active applications in an optimal way.

• Second, the OS has to support platforms (e.g. TI OMAP and ST Nomadik [1, 2]) which consist of a large number of heterogeneous processing elements (PE), each with its own set of capabilities. These platforms combine the advantages of parallel computing on multiple processors with single-chip integration of SoCs. They provide high computational performance at a low energy cost, where as typical embedded systems (e.g. handheld devices such as Personal Digital Assistants (PDAs) and smartphones) are limited by the restricted amount of processing power and memory. As the application complexity grows, the major challenge is still the right parallelisation (both data level and functional level, both coarse grain and fine grain) of these applications and their mapping on the MP-SoC platform.

• Third, the PEs in the platform communicate with each other independently and concurrently. Traditional shared medium communication architectures (e.g. buses) cannot support the massive data traffic. A flexible interconnect Network-on-Chip (NoC) [3, 4] must be adopted to provide reliable and scalable communication [5, 6].

Growing SoC complexity makes communication subsystem design as important as computation subsystem design [7]. The communication infrastructure must efficiently accommodate the communication needs of the integrated computation and storage elements. In application domains such as multimedia processing, the bandwidth requirements are already in the range of several hundred Mbps and are continuously growing [8]. In switched NoCs, switches set up communication paths that can change over time, and run-time channel and bandwidth reservation must be supported by the OS. Designing such an NoC becomes a major task for future MP-SoCs, where the communication cost is becoming much larger than the computation cost. A large fraction of the timing delay is spent on the signal propagation on the interconnect, and a significant amount of energy is also dissipated on the wires. Therefore an optimised NoC floorplan is of great importance for MP-SoC performance and energy consumption.

• Finally, for memory-intensive applications such as multimedia applications, the memory subsystem represents an important component in the overall energy cost. In the memory subsystem, ScratchPad Memories (SPM) are used [9, 10], as they perform better than caches in terms of...
energy per access, performance, on-chip area and predictability. However, unlike caches, SPMs require complex design-time application analysis to carefully decide which data to assign to the SPM and software allocation techniques [11, 12]. Design-time allocation loads the SPM once at the start of the application execution, whereas run-time allocation changes the SPM contents during the execution to reflect the dynamic behaviour of the application.

To alleviate the OS in its run-time decision-making, and to avoid conservative worst-case assumptions, a very intimate link between the application layer and the platform layer is required. To this end, we have shown in previous work [13, 14] that it is better to add a customised ultra-lightweight run-time management layer on top of the basic OS services. This layer was developed for scheduling concurrent tasks on embedded systems. It was intended to optimise the energy consumption while respecting the application deadlines only. Our approach is currently extended in the MP-SoC context to map the applications on the platform, according to application constraints (e.g. performance, user requirements) and available platform resources. Our extended approach consists of both the following phases:

1. It selects in a predictable way a mapping from its Pareto set, according to the available platform resources, in order to minimise the total energy consumption of the platform, while respecting all application constraints.
2. It performs Pareto point switches (Fig. 2, restricted to two dimensions), that is it assigns the platform resources, adapts the platform parameters, loads the task binaries from the shared memory in the corresponding local memories and issues the execution of the code versions according to the newly selected Pareto points. When Application A starts, it is assigned to three PEs, whereas run-time allocation changes the SPM contents during the execution to reflect the dynamic behaviour of the application.

The design-time exploration phase of our approach is the main contribution of our study. The resulting Pareto set of optimal mappings, to be stored into the MP-SoC platform, and used as input for our run-time manager, is essential to alleviate the OS in its run-time decision-making, and to avoid conservative worst-case assumptions. Two representative real-life applications, an image processing algorithm and a video codec multimedia one, simulated on our MP-SoC platform simulator, are also used to illustrate the optimal trade-offs offered by the design-time exploration to the run-time manager.

2 Related work

In recent years, industrial MP-SoC components have been introduced by companies like Texas Instruments and ST Microelectronics. Current OSs like the TI DSP/BIOS kernel with the DSP/BIOS link, the Quadros RTXC RTOS and the Enea Systems OSE RTOS are clearly focused on low-level run-time management (i.e. multiplexing the hardware and providing uniform communication primitives). They only provide an abstraction layer on top of the hardware, they expand and link together existing technologies, but they are not designed for the emerging MP-SoC environment. Support for SPMs, NoCs, dynamic power management, QoS-aware and application-specific run-time management, is lacking. Hence, none of these existing RTOS represents the ideal glue layer for MP-SoCs. The user is supposed to implement his own run-time manager on top of the RTOS kernel.

State-of-the-art tools and design practice also are not in a shape yet to meet the needs presented previously. Currently, two diverging strategies are developed to cope with the design complexity of application-specific and heterogeneous MP-SoC platforms: either the IP-driven approach, or the design-flow-driven approach [15].

Related to the IP-driven approach, to reuse cores on different MP-SoC platforms, a single standard interface

---

**Fig. 1** Pareto set generated by our design-time exploration

**Fig. 2** Pareto point switch
should be agreed upon. Several innovative design methodologies have recently appeared:

- The approach developed at Philips, Eindhoven is based on a task-level interface [16], called TTL. This is used both for modelling concurrent applications and as a platform interface for integrating hardware and software cores on MP-SoC platforms. As static task graph models are no longer sufficient, new reconfiguration services have been introduced in [17]. The underlying inter-task communication protocol is presented in [18].
- However when the application imposes performance constraints and cost optimisations, a design customisation is required in addition of the pre-designed component integration. To solve this issue, the approach developed at TIMA, Grenoble, proposes a unified model [19] to represent all interfaces for SoC design. This model, called Service Dependency Graph (SDG), describes interactions between hardware and software as services, and it enables full system simulation at different abstraction levels. The associated design methodology based on a two-layer hardware-dependent software (HdS), generated from the SDG, is described in [20].
- Other academic (Metropolis, SysExplorer) and commercial tools (CoWare ConvergenC, Summit Design Visual Elite) also exist supporting modelling and analysis of SoCs.

In these IP-driven approaches, synthesis is still component-centric and has no integral view on the entire system on the MP-SoC platform. Each application is synthesised separately without considering the interacting data-dependent tasks and the shared global resources. Scheduling the accesses to the global resources is determined by local decisions. The designer has to ensure manually that the system communicates properly via buses and shared memories.

Related to the design-flow-driven approach, several global optimisation issues are considered in the academic world: application parallelisation, task scheduling and dynamic reconfiguration. Next we focus on task scheduling and dynamic reconfiguration, for which our design-time exploration offers trade-offs.

2.1 Task scheduling

Task scheduling is needed to guarantee application performances and platform resource constraints, and to optimise objectives such as response time, energy consumption and battery lifetime enhancement [21]. For distributed systems [22], it has the following characteristics. First, it assumes an homogeneous architecture. Second, the most important optimisation objective is performance. Third, the focus is mainly on computation rather than on communication. For MP-SoC platforms, task scheduling becomes more complicated [23]. Its impact on the energy consumption becomes more significant, whereas performance is a constraint to be satisfied. In this context, task scheduling mainly consists of the following actions: (a) spatial mapping, determining on which processor a task must be executed, (b) temporal mapping, deciding the order in which those tasks are executed, (c) communication routing, both spatial and temporal, (d) dynamic voltage/frequency scaling (DVS/DFS), determining the processor supply voltage and clock frequency if it is allowed and (e) dynamic power management (DPM), either shutting down or swapping platform components into some sleep mode, whenever these enter an idle state.

Scheduling algorithms can be divided into design-time (also called static or off-line) and run-time (also called dynamic or on-line) algorithms. However at design time, it is unknown which applications are simultaneously active and which resources are available on the platform. So, a design-time scheduling cannot solve the problem efficiently.

Energy consumption is increasingly an issue not only for battery-operated devices. Even if unlimited power is available, a large number of components tightly packed onto a chip poses cooling and reliability problems. An important way to reduce the energy consumption is to shut down or slow down functional components which are idle or under-utilized, by combining DVS with DPM. A survey of system-level design techniques can be found in [14] and [24], respectively. The energy management problem for real-time applications decomposed into concurrent tasks and implemented on MP-SoC platforms with DVS is already addressed for several years. The most recent scheduling approaches, combining application mapping and DVS, can be found in [13, 25–28].

Other approaches combine application mapping with some run-time communication management. Reference [29] evaluates a run-time spatial and temporal mapping algorithm. This finds the right processor for a certain task and the appropriate communication path, using a library of different implementations per application (for an ARM, or a DSP, or an FPGA). It minimises the global energy consumption, while guaranteeing all application performances and platform resource constraints. The application is mapped when it is started. However, when certain events happen, the mapping might be reconsidered and/or the communication links might be re-routed. Reference [30] presents a unified single-objective algorithm, coupling spatial mapping of tasks, spatial routing of communication and temporal mapping assigning time slots to these routes. The real-time communication requirements are considered to guarantee that application performances are met. Reference [31] introduces a novel energy-aware algorithm which statically schedules application-specific communication transactions and computation tasks onto heterogeneous NoC architectures.

2.2 Dynamic reconfiguration

Multimedia applications are becoming more complex, and multiple use cases and optimal mappings need to be supported. Moving from one mapping to another is the result from user interactions or changes in the platform resource availability when new applications are activated. Adapting the mapping of an active application is called dynamic reconfiguration, or task migration. The key challenge is of course to maintain the real-time behaviour and the data integrity of the overall set of active applications. Nevertheless, dynamic reconfiguration is a powerful mechanism to improve the MP-SoC platform utilisation, avoiding some idle computing resources, while others are overloaded.

Dynamic reconfiguration has been an important topic in distributed systems [32–37], where the main focus is to achieve transparent system maintenance and evolution. Although our context is MP-SoC platforms, the distributed nature of the applications and the multitude of processors share many characteristics with distributed systems. Some common issues are task state representation and message consistency during reconfiguration [38].

For MP-SoC platforms, following issues also receive attention. Reference [39] supports incremental
reconfiguration, at the level of tasks, ports and channels. Reference [40] offers high reconfiguration freedom at the communication level. Reference [41] and [42] present different reconfiguration interfaces. Reference [43] and [44] propose efficient run-time support for reconfiguration of inter-task communication and for resource management, that is, workload sharing and admission control, to improve the platform resource utilisation, while preserving all timing constraints. To perform task migration, tasks are often assumed to be suspendable at any time, which can be difficult to achieve, and possibly for significant periods of time, which is unacceptable in real-time behaviour. Reference [45] introduces a technique to migrate tasks without suspending them.

Other important issues for both task scheduling and dynamic reconfiguration in MP-SoC platforms are: (a) the design-time application mapping and platform exploration, and (b) the run-time decision-making for the overall set of active applications, according to application constraints and platform resources. Design-time exploration allows to avoid conservative worst-case assumptions, while also eliminating large run-time overheads on the state-of-the-art RTOS kernels. Reference [46] proposes an exploration technique restricted to SoC platforms with only one processor core, and exploring SoC configurations with different power and performance trade-offs, for any application to be mapped on the SoC platform. In contrast to [46], our study is intended for MP-SoC platforms, and it proposes a combined application mapping and platform exploration approach.

3 Overview of our customised run-time management

To meet the needs presented earlier, that is, to alleviate the OS in its run-time decision-making, and to avoid conservative worst-case assumptions, we propose a customised run-time management approach to map the applications on the platform, consisting of two phases: (a) a design-time mapping and platform exploration per application; (b) a low-complexity run-time manager incorporated on top of the basic OS services. This run-time manager globally optimises costs (e.g. energy consumption) across all active applications, according to their constraints (e.g. performance, user requirements) and available platform resources. It also performs low cost switches between possible mappings of a same application, as required by environment changes.

A similar conceptual approach [13, 14] was already developed for scheduling concurrent tasks on embedded systems. However, this was intended to optimise the energy consumption while respecting the application deadlines only. Other differences are presented in the following too.

In contrast to the conventional approaches that generate only one solution for each application, the first phase of our approach is a design-time application mapping and platform exploration. For each application, this exploration generates a set of optimal mappings in a multi-dimensional design space (Fig. 1), instead of a two-dimensional one in [13, 14]. Current dimensions are application constraints (e.g. performance, user requirements), used platform resources (e.g. memory usage, number of used processors per processor type, communication bandwidth, clocks and processor supply voltage if it is allowed) and costs (e.g. energy consumption). Only points being better than the other ones in at least one dimension are retained. They are called Pareto points. The resulting set of Pareto points is called the Pareto set. This design-time exploration phase is the main contribution of our study. It is detailed in later on.

Dependent on the application constraints, and on the availability of the platform resources, any one of these Pareto points, representing application mappings, will be best selected by the run-time manager. Unlike [13, 14], each Pareto point is also annotated with a code version. The different code versions refer to different parallelisations of the application into parallel tasks and to data transfers (also called block transfers [47]) between SPMs and local memories. The characterisation and efficient merging of all these code versions, called Pareto-based application specification, is presented in [48].

Hence, in total, our Pareto set is made up for any application of optimal mappings characterised by a code version together with an optimal combination of application constraints, used platform resources and costs. The description of data structures storing information related to this Pareto set and the Pareto points is out of the scope of this study.

The full exploration is done at design time, whereas the critical decisions are taken during the second phase of our approach by a low-complexity run-time manager (Fig. 3). This latter provides the following services:

- Whenever a new application is activated, our run-time manager parses its Pareto set provided by the design-time exploration and stores it in the shared memory of the MP-SoC platform, including all task binaries.
- Whenever the environment is changing (e.g. when a new application/use case starts, or when the user requirements change), for each active application, our run-time manager reacts as follows. First, it selects in a predictable way a mapping from its Pareto set, according to the available platform resources, in order to minimise the total energy consumption of the platform, while respecting all application constraints. A heuristic solving this optimisation problem, and being fast enough for MP-SoC platforms, is presented in [49]. Second, it performs Pareto point switches (Fig. 2, restricted to two dimensions), as explained in earlier. The Pareto point switch technique bears some resemblance with dynamic reconfiguration. It can switch other mappings, but, in contrast to dynamic reconfiguration, it involves more complex run-time trade-offs offered by the multi-dimensional Pareto sets.

4 Demonstrator

An important contribution of this study is also the practical flow that is required for our design-time exploration,
applied to two real-life applications and simulated on our MP-SoC platform simulator.

4.1 Cavity detector application

The first driver application (Fig. 4) that this study uses is a cavity detector algorithm. This is an image-processing application used mainly in the medical field for detecting tumour cavities in computer tomography pictures [50]. The starting algorithm, expressed in C code, has one image frame of $M^*N$ pixels as input, and one as output. The main bottlenecks of this algorithm are: (a) important data transfer and storage because of large data arrays (for an image of 256 K pixels, as the one shown in Fig. 4 and used for our experiments, the internal data size is about 2 MB); (b) bad performance because of the presence of a complex loop ($M^*N$ iterations and a large body with many data dependencies).

4.2 QSDPCM application

As second driver application, an inter-frame compression technique for video images, called Quadtree Structured Difference Pulse Code Modulation (QSDPCM) is used [51]. It is representative for many of today's video codec multimedia applications. It involves a three-stage hierarchical motion estimation (ME4, ME2 and ME1), followed by a quadtree-based encoding of the motion compensated frame-to-frame difference signal, a quantization, and a Huffman-based compression (QC). Two image resolutions are allowed: either QCIF, with image size 176 * 144 pixels, or VGA, with image size of 640 * 480 pixels. In our experiments, the QCIF resolution is used.

The starting algorithm, expressed in C code too, has two image frames (the previous and current ones) as input, and one bit stream as output (Fig. 6a). It presents similar bottlenecks to the ones in the cavity detector.

4.3 MP-SoC platform simulator

Our MP-SoC platform simulator (Fig. 5) assumes a platform composed of: (a) processor nodes with local memories and local buses; (b) distributed shared memory nodes; (c) communication assists similar to direct memory Access (DMA) controllers, providing high-level services to processors and shared memories for efficient data transfers; (d) input/output (I/O) nodes; (e) a communication architecture, being the AEtheral NoC.

The main platform parameters that can be explored at present are: the network clock, the maximum number of time slots, the number of routers, the processor clock and supply voltage if it is allowed, the memory clock, the read/write communication bandwidth between a processor and a shared memory, the number of processors to be used by the application, the local memory usage and some QoS requirements (either guaranteed throughput, or best effort). Here, we use an in-house TI C62 DSP ISS to simulate processes, a network clock period of 4 ns, and guaranteed throughput as QoS requirement.

5 Design-time application and platform exploration

In our run-time management approach, presented earlier, a multi-dimensional Pareto set of mappings is generated by a design-time exploration for any application to be
mapped on the MP-SoC platform. This exploration is done in the following steps.

5.1 Application code tuning

Code tuning is a preprocessing step needed to derive a clean application specification with efficient data management and processing. Ad hoc code transformations, commonly used by designers, consist in:

- Minimising the size of internal large arrays. This can be done with the help of the MEMORYCOMPACTATION tool [52, 53] in ATOMIUM [54].
- Optimising the performance, mainly the loop performance, especially by reducing the number of clock cycles per iteration, and by achieving software pipelining. Typical transformations to improve the performance are: avoid function calls by explicitly inlining relevant functions, unroll loops with a small number of iterations, remove conditions, split loops with a large body into smaller ones and simplify modulo operations mainly in array index computations. This last transformation can be done with the help of the RACE tool [55] in ATOMIUM.

In our approach, to preserve these optimisations in later code refinements, and to derive several code versions with minimum code duplication, any optimised loop is encapsulated in a function, called kernel.

Cavity detector experiments: The internal arrays can be compacted, and for an image of 256 K pixels, this memory compaction yields a reduction of 99.6% in the internal data compacted, and for an image of 256 K pixels, this memory compaction yields a reduction of 99.6% in the internal data.

Experiments: Related to the functional-level parallelisation, the QSDPCM can be naturally partitioned into either three tasks (ME42, ME1 and QC), or two tasks (ME42 and ME1 merged with QC). To further alleviate the computation effort of ME1, the input frames can be divided into row blocks to parallelise ME1 at the data level. Up to five parallel ME1 tasks have been considered, beyond which no performance gain is reached any more because of very large task synchronisation and block transfers (BT) overhead. This is illustrated later. These QSDPCM parallelisations are illustrated in Fig. 6b.

Parallelising the cavity detector is not relevant enough and only one processor is considered.

5.3 Block transfer exploration

To optimise both performance and energy consumption in the memory subsystem, parts of data arrays stored in the SPM are copied in the processor local memory from where they are accessed multiple times [47]. These copy operations (also called BT) are performed through function calls in the application code, first to issue a BT, and next to synchronise its completion with processing. This allows to perform BTs in parallel with processing and hence to improve the application performance. This is illustrated in Fig. 7, where a BT into a copy cp_prev frame is performed in parallel with a for loop processing. This allows to reduce the waiting time for this BT completion and, in this small example, to reach a performance gain of 16 cycles per iteration.

Hence, the application tuned code must still be further refined as follows: (a) select the data arrays to be stored in SPMs; (b) explore the size of copies, needed to access these arrays, and stored in the processor local memory; (c) explore the places in the code where to insert BT calls (to preserve previous performance optimisations, these BT calls may not be inserted inside kernels). This exploration can be done with the help of the extended MHLA tool [47] in ATOMIUM. Several efficient solutions, yielding different local memory usages and performances, exist for the copy sizes and the places in the code where to insert these BT calls.

Hence, considering all combinations of BT solutions in all tasks of any parallelised application gives rise to a huge number of different application code versions. A Pareto-based specification, merging all of them, and allowing efficient loading of any task binary into the platform is required. This specification is characterised in [48].

Cavity detector experiments: The compacted internal arrays are small enough to be stored in the processor local memory. However, the arrays Image In and

\begin{align*}
\text{QSDPCM experiments:} \quad & \text{The starting code requires 45 M cycles per QCIF frame. The resulting algorithm is illustrated in Fig. 6a, where each module is a loop manipulating two pixel blocks at each iteration (the one from the current frame, and the other from the previous frame). The optimised code requires 6 M cycles per QCIF frame, yielding a performance gain of 86.6%. The code size is still negligible: 263 Kbytes.} \\
& \text{For this application, the code size is negligible: 116 K bytes after compilation with the TI C62 compiler.} \\
& \text{Processing: (16 cycles)} \\
& \text{BT / processing}
\end{align*}

5.2 Parallelisation exploration

Parallelising an application can be done both at functional and data levels. At the functional level, the algorithm is partitioned into smaller tasks, and synchronisation requirements between them are identified to allow pipelined execution of these tasks. At the data level, for instance, in video applications, images can be divided into block of rows. Any task parallelised at the data level deals with its own block of rows.
Image_out, storing the input and output images, respectively, must still be stored in the SPM. To access them, two copies are needed to allow to read (resp. write) BTs: Image_in_cp (resp. Image_out_cp).

Related to the copy sizes, two relevant possibilities emerge: (a) either the copy is able to store only one pixel line; then the BT must be completed during the current iteration, before the kernel functions start to manipulate it; (b) or the copy is able to store two pixel lines: one manipulated during the current iteration, and another used by the BT performed in parallel with this iteration, and that must be ready for the next iteration only.

Related to BT calls, the following consideration must be taken into account: (a) Image_in_cp being read in kernel1() implies that the corresponding read BT must be synchronised before calling this kernel; (b) Image_out_cp being written in kernel4() implies that the corresponding write BT can be issued only after the execution of this kernel, and that it must be synchronised before the next call of this kernel. Except these constraints, some freedom is left where to place the BT calls, and this is explored.

Five code versions are retained, with different local memory usages (storing both initial internal data arrays and copies) and BT calls (Table 1). These BTs are performed either sequentially or in parallel with processing, yielding different performances.

**QSDPCM experiments:** Three arrays (storing the current image frame, the previous one, and some internal data required in ME42()) are too large and must be stored in the SPM. Several efficient BT solutions are explored. Table 2 reports for the task ME1 the resulting processor local memory usage. Similar BT solutions are derived for the tasks QC and ME1_QC, whereas only one efficient BT solution is derived for ME42.

### 5.4 Platform exploration

In our multi-dimensional design space, dimensions related to the used platform resources are memory usage, number of used processors per processor type, read/write communication BandWidth (BW) between the SPM and the processor local memory, clocks and processor supply voltage if it is allowed. The main platform parameters that can still be explored at this step are: the processor clock period and the requested read/write communication BW. The number of used processors is fixed by the previous parallelisation exploration. The memory usage is fixed by the previous block transfer exploration.

We can observe that:

- The processor speed influences the BT waiting time as follows: (a) the more BTs in parallel with processing, the smaller the BT waiting time; (b) for any BT in parallel with processing, the slower processor, the smaller BT waiting time. Hence, the best performing code is the one with the most BTs in parallel with processing. But this has a price, which is the local memory usage with larger data copies to transfer data in advance for future processing.
- Requesting more read/write BW also allows to increase the performance, but only to some threshold from which the maximum throughput of the application is achieved.

**Cavity detector experiments:** In Fig. 8, the processor speed is explored and its influence on both BT waiting and processing times is reported. This exploration[Note 1] is performed on three different application codes: Code 0, where both read and write BTs are performed sequentially with processing, which implies that the BT waiting time is independent from the processor speed; Code 1 and Code 2 (Table 1) with larger data copies, but with BTs parallel with processing. As mentioned earlier, the processing time is only determined by the processor clock period and the number of processing cycles of the application (i.e. 28.7 per pixel). Code 2 is the best performing code. However, compared with Code 1, it needs 28.6% more local memory to be 29.3% faster (for a processor clock period of 8 ns).

In Fig. 9, the requested read/write BW to the SPM is explored[Note 2] for Code 1, and the shown Pareto points indicate the threshold from which requesting still more BW does not make sense any more.

Fig. 10 illustrates the code distribution[Note 3] in the Pareto set. Dependent on the requested read BW, the required performance of the application, and the available local memory space, any of the six codes can be best selected by the run-time manager.

**QSDPCM experiments:** In Fig. 11, the number of processors is explored[Note 4] for different read/write BW. The shown Pareto points indicate the threshold from which requesting still more BW or more processors does not make sense any more. As already mentioned no performance gain is reached any more because of very large task synchronisation and BT overhead.

### 6 Conclusion

In this study, we describe our design-time application and platform exploration that enables a customised run-time management for MP-SoCs. This exploration generates for each application a multi-dimensional Pareto set of optimal mappings. The optimal trade-offs offered by the design-time

---

**Table 1: Block transfer exploration for the cavity detector**

<table>
<thead>
<tr>
<th>Code version</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc. mem. usage (bytes)</td>
<td>12 800</td>
<td>17 920</td>
<td>15 360</td>
<td>15 360</td>
<td>15 360</td>
</tr>
</tbody>
</table>

**Table 2: Block transfer exploration for MEI task in QSDPCM**

<table>
<thead>
<tr>
<th>Code version</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc. mem. usage (bytes)</td>
<td>1540</td>
<td>1796</td>
<td>1972</td>
<td>2228</td>
</tr>
</tbody>
</table>

---

Note 1: A read/write bandwidth to the SPM of 40 MB/s is requested.
Note 2: A processor clock period of 2 ns is considered.
Note 3: For a processor clock period of 2 ns, and a requested write BW of 100 MB/s.
Note 4: A processor clock period of 2 ns is considered.
exploration to the run-time manager are illustrated on two real-life applications (image processing and video codec multimedia). After tuning the starting code, several code versions are derived trading different performances for memory and processor usage, according to the available platform resources. Our future work includes the run-time support to allow Pareto point switch and the analysis of the resulting run-time overhead.

7 Acknowledgment

This work has partly been funded by the 4S project (IST 001908), financed by the European Commission in the context of Framework Programme 6.


