Optimized Buffer Allocation in Multicore Platforms

Maximilian Odendahl*, Andrés Goens*, Rainer Leupers*, Gerd Ascheid*,
Benjamin Ries†, Berthold Vöcking† and Tomas Henriksson‡
*Institute for Communication Technologies and Embedded Systems
†Department of Computer Science I
RWTH Aachen University, Aachen, Germany
‡Huawei Technologies, Stockholm, Sweden

***Abstract*—With the availability of advanced MPSoC and emerging Dynamic RAM (DRAM) interface technologies, an optimal allocation of logical data buffers to physical memory cannot be handled manually anymore due to the huge design space. An allocation does not only need to decide between an on- or off-chip memory, but also needs to take an increasing number of available memory channels, different bandwidth capacities and several routing possibilities into account. We formalize this problem and introduce a Mixed Integer Linear Programming (MILP) model based on two different optimization criteria. We implement the MILP model into a retargetable tool and present a case study with representative data of the Long-Term-Evolution (LTE) standard to show the real-life applicability of our approach.**

I. Introduction

In many industry sectors, e.g. in telecommunication, memory allocation is typically calculated statically instead of dynamically during runtime to prevent fragmentation, non-deterministic allocation time and out-of-memory errors by design. Lately, the allocation options have grown tremendously for a number of different reasons. On the one hand, the increase of complexity in current and future communication standards leads to complex applications with an increasing number of logical data buffers. On the other hand, current multicore Systems on Chip (SoCs) not only provide the needed processing power to support several of these standards simultaneously, but also offer an increased number of physical memories, both on- and off-chip. Recent examples of such architectures include the Texas Instruments (TI) Keystone [1] series and the StarCore DSP series from Freescale [2].

Additionally, we see new possibilities to interface to Dynamic Random-Access Memory (DRAM) coming in the future. High bandwidth will be provided over multiple channels to fight the memory bottleneck of current system performance. Examples for these emerging techniques, which are scheduled for mass production in the next couple of years, include 2.5D (High Bandwidth Memory) or 3D (Wide I/O) DRAM integration [3]. Completely new approaches, such as the Hybrid Memory Cube (HMC) [4], where a single HMC unit will be able to offer more than 15x the bandwidth of a current DDR3 module, are expected to push the number of memory channels, latency and bandwidth capacities to another level. While these technologies offer a huge opportunity of accessing a large storage capacity very fast, it remains a grand challenge how to effectively allocate memory and distribute the access load in such a system. Therefore, even though static allocation used to be fairly simple work in the past, it has become prohibitively complex to find an optimized allocation manually. Consequently, an automated solution is highly desired.

As the main contribution of this paper, we approach the buffer allocation as an extension of a multi commodity flow problem [5]. This allows us to handle current and future multicore platforms with advanded communication architectures. We construct a graph, which is expanded over the time horizon, to create a Mixed Integer Linear Programming (MILP) model, finding an optimal allocation automatically for two different optimization criteria.

The rest of this paper is organized as follows: The following section gives an overview of the related work. Section III gives an intuitive description of the problem followed by a formalization of the same. A practical implementation of the formal concept is given in Section IV. Section V applies the solution and tooling to a case study, solving the allocation problem for a Long-Term-Evolution (LTE) use case. Conclusions are given in Section VI.

II. Related Work

The buffer allocation problem has emerged as a new challenge from recent progress in multicore platforms. To the best of our knowledge, no solution or approach exists in the literature. Nevertheless, using a MILP model to solve allocation problems has been used for many years during hardware and software synthesis. Authors in [6] and [7] use a binary MILP model for the allocation of multiport memories during data path synthesis. This allows to minimize hardware cost during ASIC design. More recent examples of allocation problems using MILP include [8], where the authors present a model and thorough theoretical analysis to generate an optimal distributed shared memory architecture. On related work closer to our software-centric challenge, authors in [9] also use a MILP to solve the problem of application-specific memory allocation. Their approach, however, is a more holistic one which includes a way of extracting memory accesses data automatically from C code as well as a tool for changing the source code to reflect the calculated allocation.

All of the models for the allocation impose restrictions on the supported platforms, as neither a distinction between different paths nor a distinction between different bandwidth capacities along a single path are supported. Additionally, they use a huge amount of binary variables, making it scale poorly.

In this paper, we address both limitations. We believe it is the first to support a flexible architecure model by combining

978-3-9815370-2-4/DATE14/©2014 EDAA

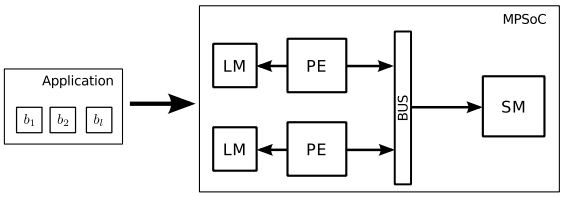


Fig. 1. Simple Mapping of Buffers

the buffer allocation problem with a multi commodity flow problem into one MILP model. Additionally, we are able to solve our model even for large input data by using an elegant construction of the constraints.

III. Buffer Allocation

Before formalizing the buffer allocation problem, a small example is given to illustrate the challenges posed in the presence of today's advanced platforms. An application consisting of a set of logical buffers is given on the left side of Figure 1. Such an application is described in an abstract way as a number of *flows*. Each flow accesses a logical buffer (b) from a processing element (PE) with a certain bandwidth requirement. While the assignment of a flow to a specific processing element is given, static buffer allocation assigns each logical buffer to a physical memory. While doing so, it must ensure that buffers fit into the available memory, bandwidth requirements are met and the load on the sytem is as small as possible.

Consider a simplified architecture given on the right side of Figure 1, consisting of two processing elements (PE), each one with a local memory (LM), and a bus connecting a shared memory (SM). Assume also that each communication possibility, represented by an arrow inside the given MPSoC, has an identical bandwidth capacity, i.e. data width and clock frequency are the same throughout the platform. Placing the logical buffers into memory for this architecture is trivial as the options are very limited. The allocation choice for a single buffer, assuming it fits into local and shared memory, is not critical to the performance of the application due to identical bandwidth capabilities to the memories. Considering the complete system, the only issue left for a designer is to manage the load of the bus, which can still be handled manually.

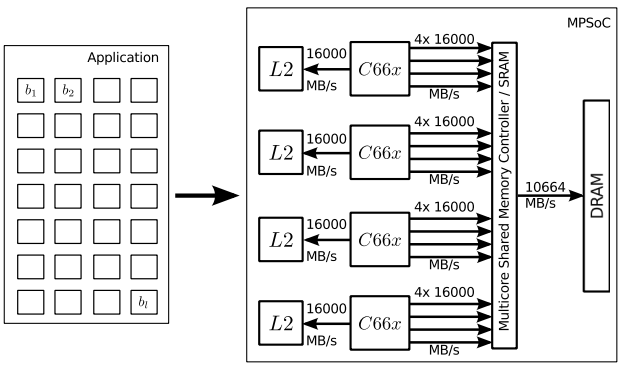


Fig. 2. Complex Mapping of Buffers to TI C6670

Now consider the realistic problem setup shown in Figure 2, which shows an increased number of logical buffers and a high-level overview of the TI C6670's architecture [1]. As can be seen, the number of allocation options has grown tremendously. We now not only have an increased number of memories, but also different levels of shared memory (L2 SRAM, DRAM) with different sizes. Additionally, the platform features different bandwidth capacities throughout the system. Obtaining an optimal allocation is already difficult for this example in which different buffer sizes and different lifetimes, which can be identified in a real application, were not even considered. Solving the entire problem is even more challenging, making an automated solution desirable to find an optimal allocation.

A. Problem Formulation

A multicore platform, such as the one given in Figure 2, consists of a set of processing elements $\mathcal{PE}$, a set of physical memories $\mathcal{PM}$ and a communication architecture. It can be modeled as a multigraph $G = (V = \mathcal{PE} \cup \mathcal{PM}, E \subseteq V \times V)$ with a bandwidth capacity function $\mathrm{bw} : E \to \mathbb{N}$, with values in bytes per time unit. An instance of the *buffer allocation problem* consists of a given *architecture graph* G, a set of *logical buffers* $\mathcal{B} = \{b_1, \ldots, b_l\}$ with their sizes $w(b_i) \in \mathbb{N}$ in bytes, an index set $\mathcal{I} \subset \mathbb{N}$ of *flows* and a set of time points $\mathbb{T} = \{1, \ldots, N\} \subset \mathbb{N}$. To each flow $i \in \mathcal{I}$, we associate a processing element $p_i \in \mathcal{PE}$, a logical buffer $b_i \in \mathcal{B}$, a *bandwidth demand* d_i in bytes per time unit between the processor and the buffer, and a *start time* $t_i^{\text{start}} \in \mathbb{T}$ and *end time* $t_i^{\text{end}} \in \mathbb{T}$, which in turn define a *lifetime* $L_i = \{t_i^{\text{start}}, \ldots, t_i^{\text{end}} - 1\}$ of the i-th flow.

A *feasible solution* to the buffer allocation problem is a mapping that assigns each logical buffer to a unique physical memory such that at any given time the size of all logical buffers mapped to a physical memory does not exceed its storage capacity $\mathrm{cap}(pm) \in \mathbb{N}$ for $pm \in \mathcal{PM}$. Additionally, each flow must route its demand d_i over one unique path from its processing element to the physical memory on which its buffer is mapped. We consider two different scenarios for interpreting a given demand:

- As a constant value per time unit that has to be sent through the network in each time point during its lifetime.
- As an absolute value, arbitrarily distributed over its lifetime. We thus set the demand that must be sent in this scenario to $\overline{d_i} := d_i \cdot (t_i^{\text{end}} - t_i^{\text{start}})$ for each flow $i \in \mathcal{I}$.

Independent of which scenario is considered, the sum of all demands going through one logical link $e \in E$ should not lead to an overload at any point in time. Moreover, we define two different optimization criteria for our definition of an *optimal solution*:

- The first criterion minimizes the maximal load among all logical links. This gives us the largest possible margin in our system to be prepared for any peaks of a flow's demand.
- The second one minimizes the maximal used memory space among all physical memories. This optimization criterion aims for the best reuse of physical memory,

allowing to reduce the memory size of a given, initial architecture as much as possible.

The difficulty lies in the fact that the routing depends on the mapping of the logical buffers and that flows can have different starting and ending times, i.e. the current state of the system changes rapidly at every time point.

If we consider only the optimal flow routing through the architecture graph, the problem can be interpreted as an integer concurrent multi commodity flow problem (ICMCF). Neglecting the buffer allocation, we are already confronted with an NP-hard problem, since ICMCF is NP-hard already for two commodities in bipartite graphs [10]. Adding the additional task of buffer allocation increases the combinatorial complexity further. As we are confronted with an NP-hard problem and we are interested in an optimal solution, we are not able to use any heuristic that runs in polynomial time. In order to solve an ICMCF instance, it is a common approach to formulate a MILP model [11]. We will present a combined MILP for the buffer allocation problem including optimal routing of flow for the two different scenarios introduced above, formulated for finding a feasible and optimal solution.

B. Time Expanded Graph

1) Construction: Our MILP model is constructed based on a graph representation. To handle the change over time, i.e. how different flows can have different concurrent lifetimes, we use a standard technique of expanding a graph over the time horizon [12]. We construct a *time expanded graph* $G^{\mathbb{T}} = (V^{\mathbb{T}}, E^{\mathbb{T}})$ consisting of *copies* G_k of the architecture graph G. A copy G_k shall represent a snapshot of a system at a given time interval, i.e. the representation of the situation in the communication network during a fixed set of consecutive time points. Let

$$g + 1 := |\{t_i^{\text{start}} \mid i \in \mathcal{I}\} \cup \{t_i^{\text{end}} \mid i \in \mathcal{I}\}|$$

be the number of distinct timepoints at which a flow starts or one has ended. Clearly, $g + 1 \leq 2|\mathcal{I}|$. We partition $\mathbb{T}$ into g disjoint sets $\mathbb{T}_k$, denoted by *groups*, between the start of a new flow or after one has ended and one time unit before the next such change. For every $1 \leq k \leq g$, we also note which flows $i \in \mathcal{I}$ are alive during $\mathbb{T}_k$ and denote the set of these as:

$$\mathcal{I}_k = \{i \in \mathcal{I} \mid \mathbb{T}_k \subseteq L_i\}$$

For each $k \in \{1, \ldots, g\}$, we create a copy G_k of the original graph and define scaled capacities for G_k by:

$$\text{bw}((e)_k) = \text{bw}(e) \cdot |\mathbb{T}_k|$$

where for each edge $e \in E$, $(e)_k$ denotes the copy of e in the k-th copy of the graph G_k. Note that strictly speaking, the bandwidth capability is converted during this step into a data capacity limit.

For each flow $i \in \mathcal{I}$, we add an artificial node s_i to the graph and join each flow node to all copies of its source:

$$\text{for all } i \in \mathcal{I},\ k \in \{1, \ldots, g\} \text{ with } i \in \mathcal{I}_k : \quad (s_i, (p_i)_k) \in E^{\mathbb{T}}$$

where $(p_i)_k$ denotes the k-th copy of the processing element node associated to the flow i. We need to add a final set of nodes which correspond to logical buffers. For this, we first define the lifetime of a buffer $b \in \mathcal{B}$ as $L_b := \{t_{i_1}^{\text{start}}, \ldots, t_{i_2}^{\text{end}} - 1\}$, where i_1 is the flow to b with the smallest starting time and i_2 that with the highest end time. To complete the construction of the time expanded graph, we consider each group k and every logical buffer b for which the group k is within its lifetime, i.e. where $\mathbb{T}_k \subseteq L_b$ holds. For each such buffer $b \in \mathcal{B}$, we add a node $\mathfrak{b}_k$, which we connect to the copy of each physical memory m_k in that group: $(m_k, \mathfrak{b}_k) \in E^{\mathbb{T}}$. The bandwidth of all these additional edges is formally set to be infinite; in practice, these vertices are not taken into account for bandwidth considerations.

2) Example: To illustrate the creation of the time expanded graph, a simple example is given. Table I shows two different flows associated to two different logical buffers. Figure 3 shows a simple architecture graph with a processor and two communication possibilities to a memory. The bandwidth capacities are annotated on the edges. Figure 4 shows the constructed time expanded graph. Based on the timing intervals, three copies of the original architecture graph are created with their respective scaled capacities.

PE	Buffer	Demand	Start	End
p^1	b^1	10	0	5
p^1	b^2	20	2	6

TABLE I
EXAMPLE FLOW DATA

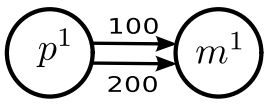


Fig. 3. Architecture Graph

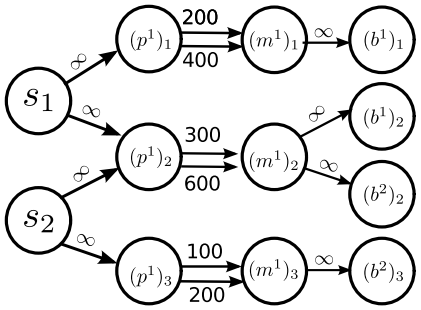


Fig. 4. Time Expanded Graph

C. Linear Programs

We now present all necessary constraints for describing our proposed MILP model to solve the buffer allocation problem.

1) Allocation Constraint: First, a linear constraint for a unique mapping of buffers to physical memories is presented. Each logical buffer $b \in \mathcal{B}$ could, in principle, be allocated to any physical memory. Hence, for each pair (b, m) of a buffer $b \in \mathcal{B}$ and a physical memory node $m \in \mathcal{PM}$, we define a binary variable $y_{b,m}$. A value of $y_{b,m} = 1$ denotes that the

buffer b is placed in the physical memory m. This leads to the following constraint:

$$\text{for all } b \in \mathcal{B}: \quad \sum_{m \in \mathcal{PM}} y_{b,m} = 1 \tag{1}$$

Since the $y_{b,m}$ are all binary, this constraint ensures that for each buffer b exactly one of the $y_{b,m}$ will be set to one.

2) Size Constraint: In every solution, the size of all logical buffers mapped to a physical memory has to be smaller than the capacity of that physical memory at any given time. Using the notation above, we get the following constraint:

$$\text{for all } k \in \{1, \ldots, g\}, \text{ for all } m \in \mathcal{PM}: \quad \sum_{b \in \mathcal{B}: \mathbb{T}_k \subseteq L_b} y_{b,m} \cdot w(b) \leq \text{cap}(m) \tag{2}$$

3) Unique Path Constraint: Every flow should send its demand over exactly one path in the architecture graph G. Let $\mathcal{P}_i$ be the set of paths starting from the processing element p_i of flow i. For each flow i and each path $P \in \mathcal{P}_i$, we introduce a binary variable $x_i(P)$, where $x_i(P) = 1$ means that flow i routes its demand via path P. For the flow $i \in \mathcal{I}$, let b_i denote the target buffer of i. The unique path constraint is specified as the following:

$$\text{for all } i \in \mathcal{I}, \text{ for all } m \in \mathcal{PM}: \quad \sum_{P \in \mathcal{P}_i:\ P \text{ ends in } m} x_i(P) = y_{b_i,m} \tag{3}$$

If a logical buffer b_i is not allocated to a physical memory m, the corresponding variable $y_{b_i,m}$ is set to zero, hence no path to this memory can be chosen. On the other hand, if $y_{b_i,m} = 1$, exactly one path will be chosen, since all $x_i(P)$ variables are binary.

An additional constraint is needed for the second scenario because for a fixed chosen path in the architecture graph, the demand $\overline{d_i}$ can be split over the corresponding paths in the time expanded graph. For example, it is now allowed that the demand $\overline{d_i}$ is routed in a single group. To formulate the constraint, we need to introduce additional variables for all demands and all paths in the time expanded graph. For each flow i and each path P' in $G^{\mathbb{T}}$ of i, let $z_i(P')$ be a continuous variable between 0 and 1. These variables indicate which percentage of the demand will be routed over the path P'. For every path P in the original graph, we define the set $\mathcal{P}_P$ as the set of all the paths from $G^{\mathbb{T}}$ corresponding to the path P. Using this notation, we are able to formulate the additional constraint for the second scenario following the same principal from constraint (3):

$$\text{for all } i \in \mathcal{I}, \text{ for all } P \in \mathcal{P}_i: \quad \sum_{P' \in \mathcal{P}_P} z_i(P') = x_i(P) \tag{4}$$

4) Bandwidth Constraint: In order to formulate the bandwidth constraint of each edge, we consider each path in $G^{\mathbb{T}}$ and its corresponding binary variable. Again, let P' be a path in $G^{\mathbb{T}}$ and let $|\mathbb{T}_{k(P')}|$ denote the number of time points in the unique copy through which P' goes. Further, let P be the path in the original architecture graph corresponding to P'. For the first scenario, we must consider the binary variables $x_i(P)$ and get the following bandwidth constraint:

$$\text{for all } e \in E^{\mathbb{T}}: \quad \sum_{P': e \in P'} \sum_{i \in \mathcal{I}: P \in \mathcal{P}_i} x_i(P) |\mathbb{T}_{k(P')}| d_i \leq \text{bw}(e) \tag{5}$$

This constraint needs to be slightly modified for the second scenario. We have to replace $x_i(P)$ by $z_i(P')$ since only a percentage of the demand will be routed through the path P' (during the times from group $k(P')$). We thus get the constraint:

$$\text{for all } e \in E^{\mathbb{T}}: \quad \sum_{P': e \in P'} \sum_{i \in \mathcal{I}: P' \in P'_i} z_i(P') \cdot \overline{d_i} \leq \text{bw}(e) \tag{6}$$

5) Objective Functions: Up to now, we have only set up constraints that guarantee any feasible buffer allocation. To obtain an optimal solution based on our two different criteria, we enhance our model with additional parameters. We will use λ_1 as a parameter for our first optimization criterion to minimize the maximal load among all logical links. We have to add λ_1^{-1} to the right hand side of constraints (5) and (6), respectively, in order for the equations to remain linear. For the first scenario, this results in the constraint:

$$\text{for all } e \in E^{\mathbb{T}}: \quad \sum_{P': e \in P'} \sum_{i \in \mathcal{I}: P \in \mathcal{P}_i} x_i(P) |\mathbb{T}_{k(P')}| d_i \leq \lambda_1^{-1} \text{bw}(e) \tag{7}$$

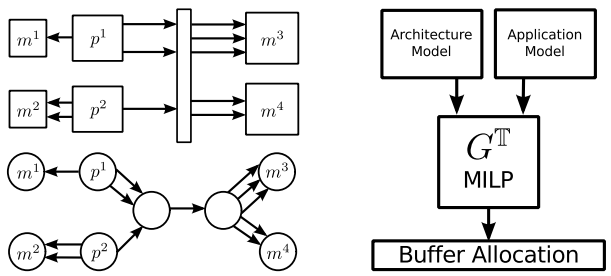
For the second scenario, we get:

$$\text{for all } e \in E^{\mathbb{T}}: \quad \sum_{P': e \in P'} \sum_{i \in \mathcal{I}: P' \in P'_i} z_i(P') \cdot \overline{d_i} \leq \lambda_1^{-1} \text{bw}(e) \tag{8}$$

Similarly, we consider another parameter λ_2 for the second optimization criterion to minimize the maximal used memory. For this case, we have to add λ_2^{-1} to constraint (2) and get:

$$\text{for all } k \in \{1, \ldots, g\}, \text{ for all } m \in \mathcal{PM}: \quad \sum_{b \in \mathcal{B}, \mathbb{T}_k \subseteq L_b} y_{b,m} \cdot w(b) \leq \lambda_2^{-1} \text{cap}(m) \tag{9}$$

Adding either the objective function $\min \lambda_1^{-1}$ or $\min \lambda_2^{-1}$ to our model gives us an optimal solution for the respective criterion. A value of $\lambda_1 > 1$ would indicate that not all logical links are loaded to full capacity, e.g. $\lambda_1 = 2$ means that all logical links are at most half loaded. The value of λ_1 gives therefore a direct hint for the available freedom of potential peaks in the bandwidth requirements of a demand. On the other hand, a value of $\lambda_1 < 1$ would mean that the problem is not feasible, i.e. some of the logical links are overloaded. Analogous to λ_1, a value of $\lambda_2 > 1$ means that the full memory capacity has not been used and the physical memory size could be reduced.



(a) MPSoC and associated architecture graph

(b) Toolflow

Fig. 5. Tooling Functionality

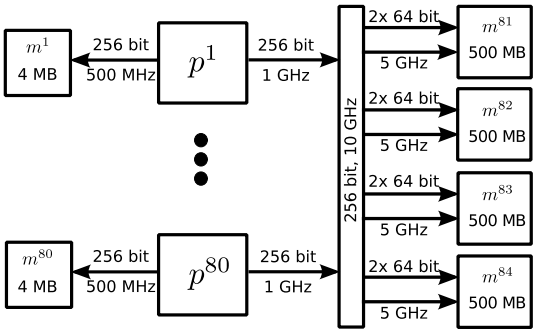


Fig. 6. MPSoC architecture

IV. Tooling

A tool was created to implement the MILP and find a solution to the buffer allocation problem. We describe the inputs to the tool and give an overview of the functionality.

A. Input

To make the tool applicable to any target MPSoC, it is designed with retargetability in mind. Therefore, an architecture model described in Extended Markup Language (XML) provides a simplified view of the target MPSoC. It describes a platform by its processing elements, memory hierarchy supporting local and shared memories and its interconnect architecture with respective access widths and clock frequencies. Using this description, diverse platforms from simple bus-based architectures to complex Network-on-Chip communication architectures can be dealt with. More than one connection between individual elements is possible, which allows several routing possibilities between different entities. This XML representation is converted automatically by the tool into our architecture graph G to ease development. Its vertices consist of individual entities of the architecture such as processing elements, memories and the communication architecture [1] . If a communication possibility exists between two elements, an edge is created with a bandwidth value $\mathrm{bw(e)}$, resulting from the given access width and clock frequency. As there can be more than one communication possibility, more than one edge can exist between two vertices. An example architecture with its associated architecture graph is shown in Figure 5 (a).

Additionally, an application model is added consisting of the input data for logical buffers and flows. They are specified as simple text files, which can be easily written manually or generated by external tools, e.g. from an automated source code analysis. As presented in Section III-A, logical buffers are annotated in the application model by their sizes, whereas a flow consists of a processing element p_i, a logical buffer b_i, a demand d_i and a lifetime L_i.

B. Functionality

Based on the timing of all flows, the tool calculates the different groups $\mathbb{T}_k$ and the lifetime of all buffers and uses a constructed architecture graph to build the time expanded graph $G^{\mathbb{T}}$ as described in Section III-B1. As a next step, the graph is used to find all possible paths from a processing element to all reachable logical buffers. With this information, all variables and constraints of the MILP can be constructed and solved using the C++ interface of the Gurobi Optimizer 5.6 [13]. Our tool includes several features for visualization of the architecture, a graphical representation of the solution and a simulation framework for the load in the communication architecture. A high level overview is given in Figure 5 (b).

[1]For certain interconnect elements, e.g. a memory bus, additional vertices are created to model a single bandwidth on a logical link. These details are not important for solving the buffer allocation problem and are thus out of the scope of this paper.

V. Case Study

A. Experimental Setup

We tested our implementation with representative data derived from a Long Term Evolution (LTE) application with a frame duration of 1 ms, resulting in 10^6 time points. The test data consists of 2709 flows accessing 907 logical buffers which need to be mapped to physical memories. The buffer sizes are in the range from 224 bytes to 50 MB, whereas the demands are in the range from 1 MB/s to 6181 MB/s. We use a hypothetical, but representative architecture consisting of 80 processing elements, each one with a local memory, and four shared memories connected by an interconnection network with an aggregate bandwidth of 2560 Gbit/s. Each shared memory can be accessed by two different connections with identical bandwidth capacities. The used architecture is shown in Figure 6. Memories are annotated by their sizes, whereas data width and clock frequency are given for connections.

B. Results

Application	Graphs & MILP	Solver
Scenario 1 Feasible	33 min	21 min
Scenario 1 Load	33 min	4 hours
Scenario 1 Memory	33 min	24 min
Scenario 2 Feasible	26 min	22 min
Scenario 2 Load	26 min	7 days
Scenario 2 Memory	26 min	32 min

TABLE II
Implementation Metrics of LTE case study

We have run our case study six times, three for each scenario. For the first run, any feasible solution is valid, whereas for the second and third run, we use our two different optimization criteria, i.e. minimizing the maximal load and minimizing the maximal used memory. A performance

evaluation of individual parts of the implementation for the six different executions of the case study are given in Table II to get an overview of the scale of our case study. In the second column, we show the combined time for calculating the groups, building the architecture and time expanded graph and for the construction of the MILP model. For our case study, this means calculating 4654 groups, constructing the architecture graph consisting of 167 nodes and 171 edges and constructing the time expanded graph consisting of 2.916.974 nodes and 183.417.566 edges. Additionally, the time for the Gurobi Optimizer to solve the model is presented in the third column. All runs were performed on a host personal computer using 4 Intel(R) Xeon(R) CPU cores running at 3.33 GHz each. The available memory of the machine is 145 GB. Note that the time neccessary to solve the model using the first optimization criterion takes much longer. This is the expected behavior, as the amount of edges in our graph is considerably larger than the amount of memories. Also note that for the two optimizing executions of scenario 2, the results of scenario 1 were taken as initial values to improve the execution time.

Application	λ_1	λ_2
Scenario 1 Feasible	1.52	1.28
Scenario 1 Load	2.45	1.22
Scenario 1 Memory	1.04	1.50
Scenario 2 Feasible	1.02	1.07
Scenario 2 Load	3.02	1.09
Scenario 2 Memory	1.02	1.63

TABLE III
MILP RESULTS

Table III presents the resulting variables. For each execution, we present λ_1 and λ_2 to be able to compare the different solutions (Note that we also obtain the allocation of buffers and assigned routing from the variables of our MILP model). It can be seen nicely how the resulting $\lambda_{1,2}$ values reflect the chosen scenario and the selected optimization criterion. For example, the largest λ_1 corresponds to the load optimized execution for the second scenario. As there is more freedom for the second scenario to shift around the demands during the time points, we expect a higher λ_1 compared to the first scenario (and longer runtime).

Comparing the result obtained from our tooling is difficult, as no other approach or an heuristic guaranteeing a feasible solution exists. We therefore implemented a second approach using a simple heuristic for the first scenario. This allows to compare the resulting $\lambda_{1,2}$ values with our solution based on the full MILP model. The heuristic starts by putting all buffers which are accessed only from one processor into its corresponding local memory (if its capacity is not yet exceeded). As a second step, each remaining logical buffer is put into the least-loaded shared memory. This heuristic leads to a reduced MILP model (i.e., with a fixed memory allocation) which is solved finding the best solution according to our first optimization criterion. The results are presented in Table IV.

Application	Runtime	λ_1	λ_2
Scenario 1	36 min	1.02	1.21

TABLE IV
HEURISTIC RESULTS

As expected, the simple heuristic yields considerably worse values for λ_1 and λ_2 than the full MILP approach. It is neither optimized for potential peaks of a demand nor able to save physical memory. Especially note that this simple approach does not guarantee a valid solution, even though in this case a feasible solution was found.

VI. CONCLUSION

In this paper, we introduced the problem of static allocation of logical buffers to physical memories. This has become a challenging task due to new abundant possibilities resulting from advanced MPSoC and memory architectures. We presented a problem formulation, MILP model and associated, retargetable tooling to solve the buffer allocation problem while taking memory footprint and bandwidth requirements into account. A solution was found completely automated in a considerable amount of time even for a large number of flows. We considered two different optimization criteria and two different interpretations of a demand. A case study using test data from the LTE standard showed the applicability of our approach.

In the future, we plan to extend the existing framework by adding caches and DMA to the architecture as well as supporting latency constraints for flows. Additionally, we will look into extending the buffer allocation problem to generate an optimized architecture for a given set of flows.

ACKNOWLEDGMENT

This work has been supported by the UMIC Research Centre, RWTH Aachen University.

REFERENCES

[1] "C6670 Multicore Fixed and Floating-Point Digital Signal Processor," [Online] Available http://www.ti.com/product/tms320c6670 (accessed 02/2013).
[2] "Freescale Multicore Starcore DSP," [Online] Available http://www.freescale.com/webapp/sps/site/taxonomy.jsp?code=MULTICORE_DS%P (accessed 09/2013).
[3] "JEDEC: 3D-ICs," [Online] Available http://www.jedec.org/category/technology-focus-area/3d-ics-0 (accessed 09/2013).
[4] "Hybrid Memory Cube Consortium," [Online] Available http://www.hybridmemorycube.org (accessed 09/2013).
[5] L. R. Ford *et al.*, "A suggested computation for maximal multi-commodity network flows," *Management Science*, vol. 5, no. 1, pp. 97–101, 1958.
[6] K. Seo *et al.*, "Allocation of Multiport Memories in ASIC Data Path Synthesis," in *International Symposium on Circuits and Systems*, vol. 1, 1994, pp. 49–52.
[7] J.-M. Jou *et al.*, "Multiport Memory based Data Path Allocation focusing on Interconnection Optimization," in *International Symposium on Circuits and Systems*, vol. 1, 1994, pp. 45–48.
[8] S. Meftali *et al.*, "An Optimal Memory Allocation for Application-specific Multiprocessor System-on-Chip," in *Proceedings of the 14th International Symposium on System Synthesis*, 2001, pp. 19–24.
[9] O. Ozturk *et al.*, "An Integer Linear Programming based Approach to Simultaneous Memory Space Partitioning and Data Allocation for Chip Multiprocessors," in *Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, 2006.
[10] S. Even *et al.*, "On the Complexity of Time Table and Multi-commodity Flow Problems," in *16th Annual Symposium on Foundations of Computer Science*, 1975, pp. 184–193.
[11] C. Barnhart *et al.*, "Integer Multicommodity Flow Problems," in *Integer Programming and Combinatorial Optimization*. Springer, 1996, pp. 58–71.
[12] L. R. Ford *et al.*, "Constructing Maximal Dynamic Flows from Static Flows," vol. 6, no. 3. INFORMS, 1958, pp. 419–433.
[13] I. Gurobi Optimization, "Gurobi Optimizer Reference Manual," 2013. [Online]: http://www.gurobi.com