# Exploring the Placement of Memory Controllers in Manycore Processors: A Case Study for Intel SCC

Patrick Eitschberger
Jörg Keller, Frank Thiele
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
firstname.lastname@fernuni-hagen.de

Christoph Kessler
Dept. of Computer and Information Science
Linköpings Universitet
58183 Linköping, Sweden
christoph.kessler@liu.se

## ABSTRACT

We present an approach to find application-specific optimal positions for memory controllers within the on-chip network of a manycore processor. The approach uses benchmark applications to represent target load, and considers computational load balance, task-to-task communication, and task-to-memory communication to model performance for a particular memory controller configuration, where the relative priorities of those criteria can be chosen by the user. A simulated annealing approach is used to find the best memory controller configuration. We use the on-chip network of the Intel SCC as a test case and find that for several applications, the positions of the memory controllers are different from their real positions. The approach can be extended to also determine the appropriate number of memory controllers, i.e. the minimum number to achieve a certain performance level.

## 1. INTRODUCTION

Manycore processors are on their way to become the standard platform in computing systems, as a consequence of the so-called power wall that makes further increase of operating frequency infeasible because of power and heat considerations. Manycore processors comprise the cores, an on-chip network, and often on-chip memory controllers (MC) to connect the manycore chip directly to one or several banks of SDRAM memory. The on-chip network serves a double purpose: to route core-to-core communications, and core-to-memory communications. In order to fulfill this task in the best possible way, the number and placement of the memory controllers is a crucial design consideration.

In order to find an optimal or close to optimal MC placement, automatic design space exploration can be used. Design space exploration requires, in particular, a cost model that typically is either analytic or based on simulation. The former method has the disadvantage that simplifications are inevitable, such as about the communication structure of applications, and thus is a rather general method. For example, trying to minimize the sum of the distances from cores to memory controllers is possible, but implicitly assumes that an application is mapped in a way that all cores have the same amount of core-to-memory communication, while the influence of core-to-core communication is ignored completely. The latter approach has the disadvantage of very long runtime, especially if real-world applications are used, so that only a restricted number of configurations can be tested.

We propose to combine the advantages of both approaches: higher speed than simulations, in order to explore many configurations, but more accuracy than simplifying analytical models. We achieve our goal by considering a set of parallel task-based benchmark applications, of which the communication structure is predictable and known. For a given placement of the memory controllers, we map the tasks in a way that balances computational load on the cores, core-to-core communication, and core-to-memory communication, with the help of an integer linear program formulation that might take latency and/or bandwidth of communications into account, and lets the user prioritize the different criteria to get an optimal solution to a multi-criterion optimization problem. This optimization is used as a sub-routine in either an integer linear program to optimally position the memory controllers, or a simulated annealing optimization that moves the memory controllers and assesses the new placement with the help of the sub-routine, until a locally or globally optimal positioning of the memory controllers is found. Currently, different numbers of memory controllers are compared by several runs of the above procedure, however the simulated annealing approach might also be extended to add or remove a memory controller in a fashion similar to genetic algorithms.

Xu et al. [9] investigate a similar problem, yet they only target core-to-memory communication and neither consider computational load nor core-to-core communication. They rate their designs with average hop count and use benchmark applications but evaluate with simulations. Abts et al. [1] use also simulations to find a best placement of the MCs. They consider only core-to-memory communication, and concentrate on improving the routing algorithm to route requests and replies differently to avoid hot spots. Awasthi et al. [2] assume a fixed memory controller placement but

*MCC'13,* November 25-26, 2013, Halmstad, Sweden.

present an optimized assignment of data to memory controllers instead of changing the placement. Our experiments indicate that the runtime of the subroutine (several seconds) is much shorter than a simulation. Yet in contrast to purely analytic solutions, we take benchmark applications' structures into account. Thus we believe that our approach combines the best of both previous approaches.

We test our proposal with the Intel SCC [5], a manycore processor with 48 cores, an on-chip mesh network, and four on-chip memory controllers located on the border of the grid. As the on-chip communications in Intel SCC are more latency-constrained than bandwidth-constrained (as the cores are rather old Pentium designs while the on-chip network and the memory controllers were specifically designed for this chip) we restrict our approach to modeling latency by distance. We find that the current placement, which might have been dictated by technological constraints, could be improved with respect to performance by moving the memory controllers to a neighboring position. We also find that an increase in the number of memory controllers improves performance, although not for reasons of increased bandwidth, but for shorter distances.

The remainder of this article is structured as follows. In Section 2, we briefly summarize facts about the Intel SCC. In Section 3, we discuss how to model computational and communication loads of applications in linear programs, and present our optimization method. In Section 4, we present our experiments, and in Section 5 we draw some conclusions and give an outlook onto future work.

## 2. INTEL SCC

The Single-Chip Cloud Computer (SCC) [5] is an experimental processor created by Intel Labs. It consists of 48 independent cores that are organized in 24 tiles. The tiles are interconnected by an on-chip high performance mesh network that also connects them to four DDR3 memory controllers to access an off-chip main memory.

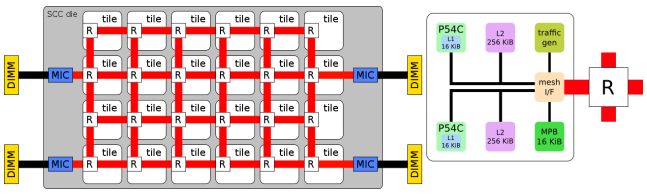


**Figure 1: SCC: overall organization and tile.**

Fig. 1(left) provides a global schematic view of the chip. The tiles are linked through a $6 \times 4$ mesh on-chip network. Each tile (cf. Fig. 1(right)) contains two cores as well as a common message passing buffer (MPB) of 16KiB (8KiB for each core). The MPB is used for direct core-to-core communication. The communication is realized via non-adaptive X-Y-routing. Thus the overall distance between communicating cores or between cores and memory controllers corresponds to the number of hops. The cores are IA-32 x86 (P54C) cores which are provided with individual L1 and L2 caches of size 32KiB (16KiB code + 16KiB data) and 256KiB, respectively, but no SIMD instructions. Each link of the mesh network exhibits a 4 cycles crossing latency including the routing activity and is 16 bytes wide.

The overall system supports a maximum of 64GiB of main memory accessible through the four DDR3 memory controllers. The memory controllers offer an aggregated peak bandwidth of 21GB/s [5]. For each core a private memory domain is assigned in the main memory, whose size depends on the total memory available (682 MiB in the system used here). Six tiles (12 cores) share one of the four memory controllers to access their private memory. The remaining part of the main memory is shared between all cores; its size can vary up to several hundred megabytes. Note that private memory is cached on cores' L2 cache but caching for shared memory is disabled by default in Intel's framework RCCE. The SCC offers no coherency among the cores' caches to the programmer when caching of shared memory is enabled. This coherency must be implemented through software methods, e.g. by flushing caches.

There are two ways to program the SCC: a baremetal version for OS development, and using Linux. In the latter setting, each core runs its own Linux kernel instance, where any Linux program can be executed. For parallel programming, Intel provides the RCCE library which contains MPI-like routines to synchronize cores and allows them to exchange messages via the MPB.

## 3. NEW APPROACH

We consider streaming-task based applications where all tasks are initially generated and concurrently run to termination. The tasks communicate with each other and access off-chip memory. Thus the application can be represented as a directed graph $G = (V, E)$, where the nodes represent the tasks, and the edges the communications. Each node is annotated with a computational rate that models the corresponding task's computational requirements. Each edge is annotated with a communication rate that models the sending task's bandwidth requirement. The communication rates can represent constant or average bandwidth. Note that even in the case that only latency is considered, the bandwidth can be used to represent this communication link's importance. To model memory accesses, each task that accesses memory is assigned a second task called *co-task*. The edges between a task and its co-task represent the bandwidth of the task's memory reads and writes respectively. Note that we only consider data access here, but not access to load program code. The co-tasks have computational rate 0. As an example, the task graph of a 3-level binary merge tree from a sorting application is depicted in Fig. 2.

The underlying machine is represented as a graph $H = (V', E')$ as well that models the on-chip interconnection network, where the nodes represent the routers and cores, and the edges the communication links. The memory controllers could be represented as extra nodes, yet normally it is convenient to have them represented by the router to which they are attached. The distance to the MC then is constantly reduced by 1, which is irrelevant for optimization purposes.

The mapping of the application can be understood as a graph embedding [4], where the application graph is the guest graph whose nodes are mapped to the host or machine

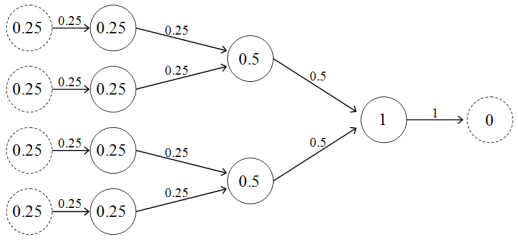


**Figure 2: Task graph for a 3-level binary merge tree. Co-tasks are indicated by dashed circles. Edge rates are averaged over time.**

graph by $m : V \to V'$. The only restriction is that co-tasks can only be mapped to nodes in the machine graph that also represent a memory controller. The classical terms of load, dilation[1] and congestion[2] can then be used to reason about the quality of this embedding. For example, the tasks from the example in Fig. 2 might all be mapped onto one core that is neighboring a memory controller, where all the co-tasks are mapped. In this case, task-to-task communication is minimized, as it can all be done without using the on-chip network (if we assume some on-chip buffer such as the MPB in Intel SCC). The latency for task-to-memory communication is minimized as well, however the memory requests are all directed to one memory controller, while the others sit idle. However, the load is spread in the most uneven way: all cores except one sit idle. If we prefer to spread the load evenly, we map the tasks of each level onto one core, which results in all communications going via the on-chip network.

Let $d(e)$ denote the dilation of edge $e$ with rate $w_e$ from the task graph, and let $l(c)$ denote the load of core $c$, i.e., the sum of all tasks' loads mapped to $c$. Then we can define

$$\begin{aligned} ld_{max} &= \max_{c \in V'} l(c) \\ com_{task} &= \sum_{e=(u,v)\in E,\ \text{both } u,v \text{ are tasks}} d(e) \cdot w_e \\ com_{mem} &= \sum_{e=(u,v)\in E,\ \text{one of } u,v \text{ is co-task}} d(e) \cdot w_e \end{aligned}$$

The quality of a certain mapping $m$ can be expressed by

$$q(m) = \varepsilon \cdot ld_{max} + (1-\varepsilon)(\zeta \cdot com_{task} + (1-\zeta) \cdot com_{mem}) \quad (1)$$

where $\varepsilon, \zeta \in [0; 1]$ represent the user's priorities with respect to the mapping. If $\varepsilon$ is close to 1, then balancing the computational load has highest priority, while $\zeta$ represents the relative priorities of the two communication types. As in Intel SCC communication is more dominated by latency than by bandwidth [6], and latency correlates with distance [6], we optimize communications by minimizing their latency. The bandwidth could be considered in a similar way, e.g. by computing the maximum over the host graph's edges' congestions. Note that $com_{mem}$ resembles the sum of distances between cores and controllers, as used in [9], yet here we build distances to tasks, weight them with their bandwidth, and also consider task-to-task communication. Note further that balancing the load minimizes the power consumption in real-time scenarios, because then the deadline can be met with lowest processor frequencies.

Our goal is to find a mapping $m^*$ such that $q(m^*)$ is minimal over all possible mappings. Assume that we can compute such a mapping (see below) for given number and positions of memory controllers. We define a *valid* memory controller configuration $mc$ as one where all memory controllers sit on distinct positions of the host graph. Furthermore, and in contrast to Xu et al. [9], we only place memory controllers at the border of the grid. This has two reasons. First, it might be technologically difficult to place memory controllers, which are connected to the chip's pins, in the mid of the die. Second, the routers get simpler as the border routers have free links anyway to connect the controllers. However, this restriction can be removed for target architectures where our objections are not relevant. The *neighborhood* $n(mc)$ of a configuration $mc$ is the set of all valid configurations $mc_i$ where one controller from $mc$ is moved to a neighboring node of the host graph. More complex neighborhoods are possible, where e.g. several memory controllers are moved at once, or where controllers can be added or removed to also determine the optimal number of controllers.

The definition of a neighborhood allows to apply a steepest-descent optimization: we determine the neighbor configuration $m_{i^*}$ with the best quality, i.e.

$$i^* = \text{argmin}\{m(mc_i) \ : \ mc_i \in n(mc)\} \ ,$$

and continue with this configuration if its quality is better than that of $mc$, until no further descent is possible. In order to escape local minima, we apply a simulated annealing approach, i.e. with a certain probability the decreases geometrically over time, we use the best neighboring configuration even if its quality is worse than that of $mc$.

To compute an optimal application mapping for a configuration, we construct an integer linear optimization which we solve with the gurobi solver (plus AMPL as frontend): To minimize the target function (1) for given $\varepsilon$ and $\zeta$, we use binary indicator variables $x_{t,c}$ where $t$ is a task and $c$ a core, and $x_{t,c} = 1$ iff task $t$ is mapped to core $c$. The constraint

$$\sum_c x_{t,c} = 1$$

ensures each task $t$ to be only mapped once. The constraint

$$\sum_t l(t) \cdot x_{t,c} \leq ld_{max}$$

for each core $c$, where $l(t)$ is the constant computational rate of task $t$ from the task graph, ensures that the computational load is balanced evenly. For $com_{task}$ and $com_{mem}$ we use similar constraints, where the distance function is modelled with the help of auxiliary binary indicator variables. Note that instead of a combination of simulated annealing and linear program, we could extend the linear program to find optimal memory controller positions, and we could use a heuristic instead of an ILP-solver for large problem instances. Details about the integer linear programs, the parameters of the simulated annealing, and the optimization runtimes can be found in [8].

[1]Dilation of edge $(u, v)$ of $G$: length of path $(m(u), m(v))$ in $H$.

[2]Congestion of edge $f$ in $H$: weighted sum of all paths $(m(u), m(v))$ comprising $f$, where $e = (u, v) \in E$.

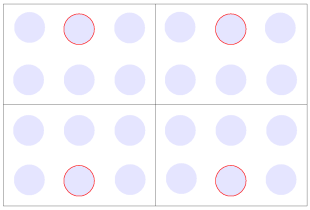

**Figure 3: Optimal SCC controller placement.**

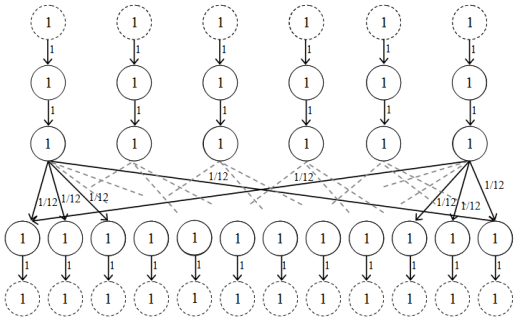

**Figure 4: Task graph for tiled MapReduce from [3].**

# 4. EVALUATION

We evaluate our approach with three benchmark applications. In all experiments we use $\varepsilon = 0.9$ and $\zeta = 0.5$; using a smaller $\varepsilon$ leads to a "collapsed" mapping onto one core as described in Sect. 3. The first benchmark is a part of parallel merge sort, where we map eight 6-level merge trees (similar to that in Fig. 2 to fully utilize the SCC (cf. [7]). We exploit symmetry and map two merge trees per $3 \times 2$-quadrant. As each tile comprises two cores, it suffices to use one merge tree as the task graph, and one quadrant with one memory controller as the host graph. The optimal result is depicted in Fig. 3. In contrast to the Intel SCC, our approach does not place the memory controller in quadrant corners, leading to a reduction of the target function by 1%. The second benchmark is tiled MapReduce [3], which leads to a task graph depicted in Fig. 4. Symmetry considerations allow an optimization on a quadrant. The optimal result looks as in Fig. 3. The third benchmark is a $4 \times 3$-grid task graph, where the 10 border tasks read from and write to memory, with computational rates 3 and communication rates 1. Here, no symmetry was applied, so that the problem was much larger than before. Also here the best configuration is like that in Fig. 3, and the target function is about 5% better than for the real placement. We also investigated configurations with 8 and 12 memory controllers. For 8 MCs, the "winning" configurations had two controllers on the middle of each side. For 12 MCs, each longer side had 4 MCs in the middle and the shorter had 2 MCs in the middle. Thus, the corners are avoided.

# 5. CONCLUSIONS

We have presented an approach to map applications optimally to a manycore processor with respect to computational load balance, task-to-task and task-to-memory communication, with user-chosen priorities. In communication, both latency and bandwidth can be considered, although we only use latency, cf. [6]. This approach is used in a simulated annealing optimization to find an optimal placement of the on-chip memory controllers. We restrict ourselves to positions at the border of the network, however the approach can also be used for arbitrary controller positions. Starting the optimization with different numbers of controllers, or extending the neighborhood function to also add or remove controllers, can be used to also find the optimal number of memory controllers. We apply this approach to evaluate the memory controller positions in the Intel SCC with the help of three benchmarks: binary merge sort, a MapReduce variant, and a grid communication application. We find that the optimal positioning in our model is different from the real configuration. Future work will comprise to test the scalability of our approach for larger networks, and to consider the effects of bandwidth considerations.

## Acknowledgements

The authors thank Intel for providing the opportunity to experiment with the "concept-vehicle" many-core processor "Single-Chip Cloud Computer". C. Kessler acknowledges partial funding by Vetenskapsrådet and SeRC.

# 6. REFERENCES

[1] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving predictable performance through better memory controller placement in many-core CMPs. In *Proc. 36th Int.l Symp. Computer Architecture (ISCA '09)*, 2009.

[2] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proc. 19th Conf. Parallel Architectures and Compilation Techniques*, 2010.

[3] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proc. 19th Conf. Parallel Architectures and Compilation Techniques*, 2010.

[4] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[5] J. Howard et al. A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling. *IEEE J. of Solid-State Circuits*, 46(1):173–183, Jan. 2011.

[6] N. Melot, K. Avdic, C. Kessler, and J. Keller. Investigation of main memory bandwidth on Intel Single-Chip Cloud Computer. In *Proc. Intel MARC3 Symposium*, Ettlingen, 2011.

[7] N. Melot, C. Kessler, K. Avdic, P. Cichowski, and J. Keller. Engineering parallel sorting for the Intel SCC. In *Proc. 4th WEPA at ICCS*, pages 1890–1899, 2012.

[8] F. Thiele. Optimization of cluster-on-chip architectures. Master's thesis, FernUniversität in Hagen, Fac. Mathematics and Computer Science, 2013. http://www.fernuni-hagen.de/imperia/md/content/fakultaetfuermathematikundinformatik/pv/mt.pdf.

[9] T. C. Xu, P. Liljeberg, and H. Tenhunen. Optimal memory controller placement for chip multiprocessor. In *Proc. 7th Int.l Conf. Hardware/software codesign and system synthesis*, pages 217–226, 2011.