Optimized On-Chip Pipelining of Memory-Intensive Computations on the Cell BE

Christoph W. Kessler
Linköpings Universitet
Dept. of Computer and Inf. Science
58183 Linköping, Sweden
chrke@ida.liu.se

Jörg Keller
FernUniversität in Hagen
Dept. of Mathematics and Computer Science
58084 Hagen, Germany
joerg.keller@fernuni-hagen.de

Abstract

Multiprocessors-on-chip, such as the Cell BE processor, regularly suffer from restricted bandwidth to off-chip main memory. We propose to reduce memory bandwidth requirements, and thus increase performance, by expressing our application as a task graph, by running dependent tasks concurrently and by pipelining results directly from task to task where possible, instead of buffering in off-chip memory. To maximize bandwidth savings and balance load simultaneously, we solve a mapping problem of tasks to SPEs on the Cell BE. We present three approaches: an integer linear programming formulation that allows to compute Pareto-optimal mappings for smaller task graphs, general heuristics, and a problem specific approximation algorithm. We validate the mappings for dataparallel computations and sorting.

1. Introduction

The new generation of multiprocessors-on-chip derives its raw power from parallelism, and explicit parallel programming with platform-specific tuning is needed to turn this power into performance. A prominent example is the Cell Broadband Engine with a PowerPC core and 8 parallel slave processors called SPEs (e.g. cf. [1]). Yet, many applications use the Cell BE like a dancehall architecture: the SPEs use their small on-chip local memories (256 KB for both code and data) as explicitly-managed caches, and they all load and store data from/to the external (off-chip) main memory. However, the bandwidth to the external memory is much smaller than the SPEs' aggregate bandwidth to the on-chip interconnect bus (EIB) [1]. For applications that frequently access off-chip main memory, such as streaming computations, stream-based sorting, or dataparallel computations on large vectors, the ratio between computational work and memory transfer is low, such that the limited bandwidth to off-chip main memory constitutes the performance bottleneck. This problem will become even more severe with the expected increase in the number of cores in the future. Consequently, the generation of memory-efficient code is an important optimization to consider for such memory-intensive computations.

Scalable parallelization on such architectures should therefore trade increased communication between the SPEs over the high-bandwidth EIB for a reduced volume of communication with external memory, and thereby improve the computation throughput for memory-intensive computations. This results in an *on-chip pipelining* technique: The computations are reorganized such that intermediate results (temporary vectors) are not written back to main memory but instead forwarded immediately to a consuming successor operation. This requires some buffering of intermediate results in on-chip memory, which is necessary anyway in processors like Cell in order to overlap computation with bulk (DMA) communication. It also requires that all tasks (elementary streaming operations) of the algorithm be active simultaneously; tasks assigned to the same SPE will be scheduled round-robin, each with a SPE time share corresponding to its relative computational load. However, as we would like to guarantee fast user-level context switching among the tasks on a SPE, the limited size of Cell's local on-chip memory then puts a limit on the number of tasks that can be mapped to a SPE, or correspondingly a limit on the size of data packets that can be buffered, which also affects performance. Moreover, the total volume of intermediate data forwarded on-chip should be low and, in particular, must not exceed the capacity of the on-chip bus.

We formalize the problem by modeling the application as a weighted acyclic task graph, with node and edge weights denoting computational load and communication rates, respectively, and the Cell processor by its key architectural parameters. We assume that only one application is using

the Cell processor at a time. Task graph topologies occuring in practice are, e.g., complete b-ary trees for b-way merge sort or bitonic sort, butterfly graphs for FFT, and arbitrary tree and DAG structures representing vectorized dataparallel computations. These applications seem to be major application areas for Cell BE besides gaming. On-chip pipelining then becomes a constrained optimization problem to determine a mapping of the task graph nodes to the SPEs that is an optimal or near-optimal trade-off between load balancing, buffer memory consumption, and communication load on the on-chip bus.

To solve this multi-objective optimization problem, we propose an integer linear programming (ILP) formulation that allows to compute Pareto-optimal solutions for the mapping of small to medium-sized task graphs with a state-of-the-art ILP solver. For larger general task graphs we provide a heuristic two-step approach to reduce the problem size. We exemplify our mapping technique with several memory-intensive example problems: with acyclic pipelined task graphs derived from dataparallel code, with complete b-ary merger tree pipelines for parallel mergesort, and with butterfly pipelines for parallel FFT. We validate the mappings with discrete event simulations. Details are given in a forthcoming paper [2].

For special task graph topologies such as merge trees, more problem-tailored solutions can be applied. In previous work [3] on pipelined parallel mergesort, we described a tree-specific divide-and-conquer heuristic and an ILP formulation to compute a good or even optimal placement of the tasks of the resulting tree-shaped pipelined computation. These results can be used to improve Cell-specific merge sort or bitonic sort implementations reported in the literature [4, 5].

In the present paper, we briefly summarize some of our very recent results [2, 3] in this area. In Sect. 2 we present optimal mapping results, and in Sect. 3 we summarize heuristic results for large task graphs. In Sect. 4, we present a new tree-specific approximation algorithm. In Sect. 5 we summarize related work, and Sect. 6 concludes.

2. Optimal Mapping of Task Graphs for On-Chip Pipelining

We start by introducing some basic notation and stating the general optimization problem to be solved. We then give an integer linear programming (ILP) formulation for the problem, which allows to compute optimal solutions for small and middle-sized pipeline task graphs, and report on the experimental results obtained for examples taken from the domain of streaming computations.

Problem definition Given is a set $P = \{P_1, \ldots, P_p\}$ of p processors and a directed acyclic task graph $G = (V, E)$ to be mapped onto the processors. Input is fed at the sources, data flows in direction of the edges, output is produced by the sinks.

Each node (task) v in the graph processes the incoming data streams and combines them into one outgoing data stream. With each edge $e \in E$ we associate the (average) rate $\tau(e)$ of the data stream flowing along e. In all types of streaming computations considered in this work, all input streams of a task have the same rate. However, other scenarios with different τ rates for incoming edges may be possible.

The *computational load* $\rho(v)$ denotes the relative amount of computational work performed by the task v, compared to the overall work $\sum_{v \in V} \rho(V)$ in the task graph. It will be proportional to the processor time that a node v places on a processor it is mapped to. In most scenarios, ρ is proportional to the data rate $\tau(e)$ of its (busiest, if several) output stream e. Reductions are a natural exception here; their processing rate is proportional to the input data rate.

In contrast to the averaged values ρ and τ, the actual computational load (at a given time) is usually depending on the current or recent data rates τ. In cases such as mergesort where the input data rates may show higher variation around the average τ values, also the computational load will be varying when the jitter in the operand supply cannot be compensated for by the limited size buffers.

For presentation purposes, we usually normalize the values of ρ and τ such that the heaviest loaded task r obtains $\rho(r) = 1$ and the heaviest loaded edge e obtains $\tau(e) = 1$. For instance, the root r of a merge tree will have $\rho(r) = 1$ and produce a result stream of rate 1. The computational load and output rate may of course be interpreted as node and edge weights of the task graph, respectively.

The *memory load* $\beta(v)$ that a node v will place on the (SPE) processor it is mapped to (including packet buffers, code, stack space) is usually just a fixed value depending on the computation type of v, because the node needs a fixed amount for its code, for buffering transferred data, and for the internal data structures it uses for processing the data. In homogeneous task graphs such as merge trees or FFT butterflies, all $\beta(v)$ are equal. In this case, we also normalize the memory loads such that each task v gets memory load $\beta(v) = 1$.

We construct a mapping $\mu : V \to P$ of nodes to processors. Under this mapping μ, a processor P_i has *computational load*

$$C_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} \rho(v),$$

i.e. the sum of the load of all nodes mapped to it, and it has *memory load*

$$M_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} \beta(v)$$

which is $1 \cdot \#\mu^{-1}(P_i)$ for the case of homogeneous task graphs.

The mapping μ that we seek shall have the following properties:

1. The maximum computational load $C^*_\mu = \max_{P_i \in P} C_\mu(P_i)$ among the processors shall be minimized. This requirement is obvious, because the lower the maximum computational load, the more evenly the load is distributed over the processors. With a completely balanced load, C^*_μ will be minimized.

2. The maximum memory load $M^*_\mu = \max_{P_i \in P} M_\mu(P_i)$ among the processors shall be minimized. The maximum memory load is proportional to the number of the buffers. As the memory per processor is fixed, the maximum memory load determines the buffer size. If the buffers are too small, communication performance will suffer.

3. The *communication load* $L_\mu = \sum_{(u,v)\in E, \mu(u)\neq\mu(v)} \tau(u)$, i.e. the sum of the edge weights between processors, should be low.

ILP Formulation We are given a task graph with n nodes (tasks) and m edges, node weights ρ, node buffer requirements β, and edge weights τ. The ILP model for mapping the task graph to a set P of SPEs is summarized in Figure 1; for more details see [2].

For a Cell with p SPEs and a general task graph with n nodes and m edges, our ILP model uses $np + mp = O(np)$ boolean variables, 1 integer variable, 2 linear variables, and $2mp + 2p + 2 = O(np)$ constraints. We implemented the ILP model in CPLEX 10.2 [6], a commercial ILP solver.

By choosing the ratio of ϵ_M to ϵ_C, we can only find two extremal Pareto-optimal solutions, one with least possible *maxMemoryLoad* and one with least possible *commLoad*. In order to enforce finding further Pareto-optimal solutions that may exist in between, one can use any fixed ratio ϵ_M/ϵ_C, e.g. at 1, and instead set a given minimum memory load to spend (which is integer) on optimizing for *commLoad* only:

$$\textit{maxMemoryLoad} \geq \textit{givenMinMemoryLoad}$$

For modeling task graphs of mergesort as introduced above, we generated binary merge trees with k levels (and thus $2^k - 1$ nodes) intended for mapping to $p = k$ processors [3]. Table 1 shows all Pareto-optimal solutions that CPLEX found for $k = p = 5, 6, 7$. While most optimizations for $k = 5, 6, 7$ took just a few seconds, CPLEX hit a timeout after 24 hours for $k = 8$ and only produced approximate solutions with a memory load of at least 37. Figure 2 shows one Pareto-optimal mapping for $k = 5$.

Solution variables:
Binary variables x, z with
$x_{v,q} = 1$ iff node v is mapped on processor q, and
$z_{(u,v),q} = 1$ iff both source u and target v of edge (u, v) are mapped to processor q.
The integer variable *maxMemoryLoad* will hold the maximum memory load assigned to any SPE in P.
The linear variable *maxComputLoad* yields the maximum accumulated load mapped to a SPE.
Constraints:
Each node must be mapped to exactly one processor:

$$\forall v \in V: \quad \sum_{q \in P} x_{v,q} = 1$$

The maximum load mapped to a processor is computed as

$$\forall q \in P: \quad \sum_{v \in V} x_{v,q} \cdot \rho(v) \leq \textit{maxComputLoad}$$

The memory load should be balanced:

$$\forall q \in P: \quad \sum_{v \in V} x_{v,q} \cdot \beta(v) \leq \textit{maxMemoryLoad}$$

Communication cost occurs whenever an edge is not internal, i.e. its endpoints are mapped to different SPEs.

$$\begin{aligned} \forall (u,v) \in E,\ q \in P: \quad & z_{(u,v),q} \leq x_{v,q} \\ & z_{(u,v),q} \leq x_{u,q} \end{aligned}$$

and in order to enforce that a $z_{(u,v),q}$ will be 1 wherever it could be, we have to take up the (weighted) sum over all z in the objective function. This means, of course, that only optimal solutions to the ILP are guaranteed to be correct with respect to minimizing communication cost. We accept this to avoid quadratic optimization, and because we also want to minimize the maximum communication load. The communication load is the communication volume over all edges minus the volume over the internal edges:

$$\textit{commLoad} = \sum_{e \in E} \tau(e) - \sum_{e \in E} \sum_{q \in P} z_{e,q} \cdot \tau(e)$$

Objective function: Minimize

$$\begin{aligned} \Lambda \cdot \textit{maxComputLoad} \ + \ & \epsilon_M \cdot \textit{maxMemoryLoad} \\ & + \epsilon_C \cdot \textit{commLoad} \end{aligned}$$

with Λ chosen large enough to prioritize computational load balancing over all other optimization goals; the positive weights $0 \leq \epsilon_M < 1$ and $0 < \epsilon_C < 1$ are chosen to give preference to *maxMemoryLoad* or *commLoad* as secondary optimization goal.

Figure 1. ILP model for mapping task graphs.

Table 1. The Pareto-optimal solutions for mapping b-ary merge trees, found with ILP, for $b = 2$, $k = p = 5,\ 6,\ 7$.

k	binary variables	constraints	*max. memory load*	*commLoad*
$k = 5$	305	341	8	2.5
			9	2.375
			10	1.75
$k = 6$	750	826	13	2.625
			14	2.4375
			15	1.9375
			20	1.875
$k = 7$	1771	1906	21	2.375
			29	2.3125
			30	2

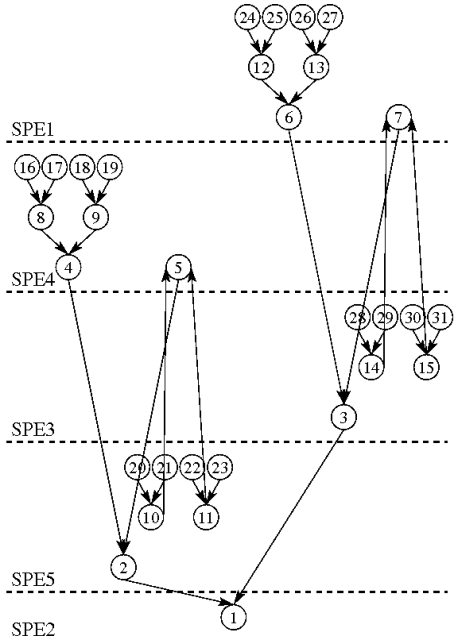

Figure 2. A Pareto-optimal solution for mapping a 5-level merge tree onto 5 processors with maximum memory load 8 (merger tasks on an SPE) and communication load 2.5 (accumulated rate of all inter-SPE edges), computed by the ILP solver.

To test the performance of our merger tree mappings with respect to load balancing, we implemented a discrete event simulation of the pipelined parallel mergesort. The simulation is quite accurate, as the variation in runtime for merger nodes is almost zero, and communication and computation can be overlapped perfectly by mapping several nodes to one SPE. We have investigated several mappings resulting from our mapping algorithm. The 5-level tree of Fig. 2 realizes a 32-to-1 merge. The maximum memory load of 8 merger tasks (needing 5 buffers each) on a SPE still yields a reasonable buffer size of 4 KB, accumulating to a maximum of 160 KB for buffers per SPE. With 32 input blocks of 2^{20} sorted random integers, the pipeline efficiency was 93%. In comparison to the corresponding merge phase in [5], memory bandwidth requirements decreased by a factor of 2.5, but as now 5 instead of 4 SPEs are utilized, this translates to a factor 1.86 in estimated performance gain. For further results for mapped merge trees, see [3].

In order to test the ILP model with dataparallel task graphs, we used several hand-vectorized fragments from the Livermore Loops and synthetic kernels, see Table 2. Such task graphs are usually of very moderate size, and computing an optimal ILP solution for a small number of SPEs takes only a few seconds in most of the cases. For two common Cell configurations ($p = 6$ as in PS3, and $p = 8$), the generated ILP model sizes (after preprocessing) and the times for optimization with memory load preference are given in Table 2. A discrete event simulation of the LL9 mapping on 6 SPEs achieved a pipeline efficiency of close to 100%. Further results and discussion can be found in [2].

3. Heuristic Algorithms for Large Task Graphs

The ILP solver works well for small task graph and machine sizes. However, for future generations of Cell with many more SPEs and for larger task graph sizes, computing optimal mappings with the ILP approach will no longer be computationally feasible. For the case of general task graphs, we developed a divide-and-conquer based heuristic [2] where the divide step uses the ILP model for $p = 2$. An example mapping is given in Fig. 3 for Livermore Loop 9.

4. New Approximation Algorithm for Mapping Merge Trees

We consider the mapping problem for task graphs with the structure of b-ary merger trees, as in b-ary merge sort. In previous work, we presented an approximation algorithm based on divide-and-conquer [3]. Its approximation guarantee for the maximum memory load mainly depends on the tree size k_0 considered as base case in the recursive solution (which is, e.g. solved optimally by the ILP method); the worst-case maximum memory load is by a factor k/k_0 larger than a straightforward lower bound (see Lemma 1), but the quality is much better in practice [3]. As an example, for $k = p = 5$ and $b = 2$ (i.e., a 32-to-1 binary merger tree), the resulting mapping μ_1 is different from the Pareto-optimal one shown in Figure 2 but has the same quality (optimal maximum memory load 8 and communication

Table 2. ILP models for dataparallel task graphs extracted from the Livermore Loops (LL) and from some synthetic kernels.

Kernel	Description	#Nodes	#Edges	ILP model for $p = 6$			ILP model for $p = 8$		
		n	m	var's	constr.	time	var's	constr.	time
LL9	Integrate predictors	28	27	333	371	2:07s	443	485	—
LL10	Difference predictors	29	28	345	384	0:06s	459	502	1:26:39s
LL14	1D particle in cell, 2nd loop	19	21	243	290	0:03s	323	380	1:05s
LL22	Planckian distribution	10	8	111	125	<0:01s	147	163	<0:01s
FIR8	8-tap FIR filter	16	22	231	299	45:04s	307	393	0:04s
T-8	Binary tree, 16 leaves	31	30	369	410	5:36s	491	536	0:11s
C-6	Cook pyramid, 6 leaves	21	30	309	400	27:56s	411	526	3:22s

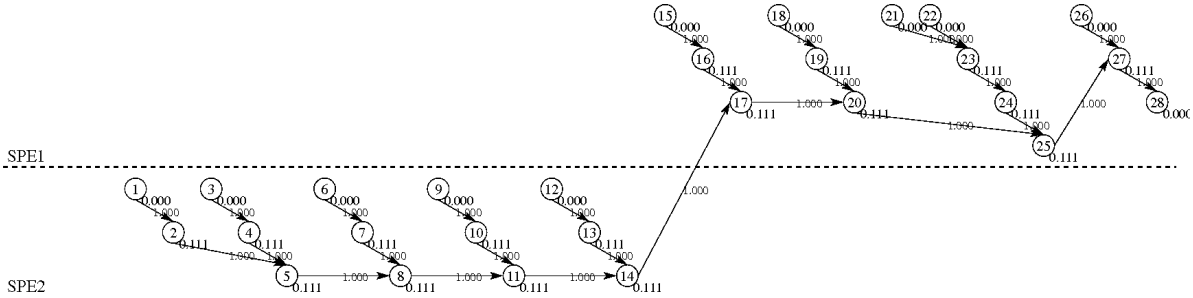

Figure 3. ILP solution for partitioning the task graph of Livermore Loop 9 into two (thus $p = 2$) subgraphs, each to be mapped separately on a Cell SPE subset of size 4.

load 2.5). The discrete event simulation reported also for this mapping a pipeline efficiency of 93%.

In the following, we give an alternative approximation algorithm where the maximum memory load is by a factor at most b larger than the lower bound, independent of the size of the tree, i.e. the number of levels.

We will use the notations from Sect. 2. Our task graph T is a complete and balanced b-ary k-level merge tree. As each task has exactly one outgoing edge, we identify the rate τ of edges with the computational load of their origin node. Thus $\tau(v, w) = \rho(v)$. As task v merges the b incoming data streams into one outgoing data stream with rate $\tau(v)$, the incoming data streams on average will have rate $\tau(v)/b$, if we assume only finite buffering within nodes. With normalization, the tree root r will have $\rho(r) = 1$, and thus each node v on level i of the tree, where $0 \leq i \leq k - 1$, has $\rho(v) = b^{-i}$ on average.

We extend ρ and τ to subgraphs of the merge tree. A subgraph's computational load is the sum its node loads, and its outgoing data rate is the aggregate rate of all edges leaving the subgraph. For example, a subtree of l levels rooted at v has computational load $l \cdot \rho(v)$ and data rate $\tau(v)$.

The mapping μ for T that we seek shall have the properties 1–3 already listed for the general case in Section 2, but in addition, it shall also fulfill:

4. As often as possible, sibling nodes (nodes u and v with a common successor w, i.e. where $(u, w) \in E$ and $(v, w) \in E$) should be mapped to the same processor.

   Note that a merger should deliver merged data buffers at an actual output rate that does not significantly fall short of the average output rate, because otherwise the preceding and subsequent mergers may be delayed, too, due to limited buffer capacity. A drop in the output rate may be caused by phases of unequal distribution of data in the input sequences, such that a merger processes, in such a phase, mainly input data coming from one subtree only, which effectively stalls the other subtree(s). Short phases can be caught by buffering (if buffers are sufficiently large) and have thus no effect, while long phases may lead to idle times on some processors. If sibling merger nodes are mapped to the same processor, such a stall of a sibling node allows to temporarily give an accordingly larger processor time share to the busier sibling(s), maintaining a more balanced overall output rate of the siblings towards the common parent node.

Lemma 1 (Lower bounds) *In any mapping μ the maximum computational load is at least k/p, and the maximum memory load is at least $(b^k - 1)/((b - 1)p)$.*

Proof: As there are b^i nodes in level i, each with com-

putational load b^{-i}, the computational load in each level equals 1, i.e. the load of the k-level tree equals k. As this load is spread over p processors, there will be at least one processor with computational load at least k/p.

As there are $(b^k-1)/(b-1)$ nodes in a k-level balanced b-ary tree, each with memory load c, the memory load of the tree equals $c \cdot (b^k-1)/(b-1)$. As this load is spread over p processors, there will be at least one processor with memory load at least $(b^k-1)/((b-1)p)$. ∎

Construction Consider as a first try the case $p = k$ and the mapping μ_0 that maps all nodes of level i onto processor P_i. Obviously, this mapping fulfils properties 1 and 3, as the computational load of each level equals 1 (see Lemma 1), and as siblings in the tree are always on the same level and hence mapped to the same processor. However, b^{k-1} nodes of level $k-1$ are mapped to processor P_{k-1}, and hence $M^*_{\mu_0} = c \cdot b^{k-1}$ and thus a factor of about $k/2 \leq k(b-1)/b \leq k$ away from the lower bound of Lemma 1. This restriction is serious, as each processor only contains a fixed amount of local memory, so that either, when we consider the memory load of each task to be fixed, the maximum number k to which this mapping scales is severely limited. If we do not fix the memory load of the task, the memory available for each node is—at least on level $k-1$, i.e. for at least half of all nodes because of $b \geq 2$—shrinking by a factor of k faster with growing k than necessary, i.e. buffer size will soon become very small, which also affects performance of data transfer.

We therefore devise a mapping μ_1 that is constructed in several steps, and in each step i maps l_i levels of the tree onto l_i processors. Let $k_0 = k$ be the number of levels and processors in step 0. In step i, if $k_i \geq 2$, we map $l_i \leq k_i - 1$ of the k_i levels, starting from the leaves, onto a respective number of processors, so that $k_{i+1} = k_i - l_i$ levels and processors remain. If $k_i = 1$, we map the tree root onto the last processor, and the mapping is complete. As each level of the tree has a computational load of 1, the mapping must be such that each processor receives a load of 1 to minimize $C^*_{\mu_1}$.

We choose l_i to be the largest power of b less than or equal to $k_i - 1$. The l_i levels then consist of $b^{k_{i+1}}$ balanced b-ary trees of l_i levels each. If $l_i \leq b^{k_{i+1}}$, then l_i divides $b^{k_{i+1}}$ because it is also a power of b, and we map $b^{k_{i+1}}/l_i$ trees on each of the processors. This balances both maximum computational and maximum memory load.

The case $l_i > b^{k_{i+1}}$ is illustrated in Fig. 4. In this case, we can write $l_i = b^x \cdot b^{k_{i+1}}$, where $x \geq 1$ is an integral number. In this case, we define $l'_i = l_i - b^x$ and first map the l'_i levels starting from the leaves. Those levels consist of $b^{k_{i+1}+b^x}$ balanced b-ary trees of l'_i levels each. As $b^x \geq x$ because of $b \geq 2$ and $x \geq 1$, it follows that $b^{k_{i+1}+b^x} \geq b^{k_{i+1}+x} = l_i$ and that this number is even an

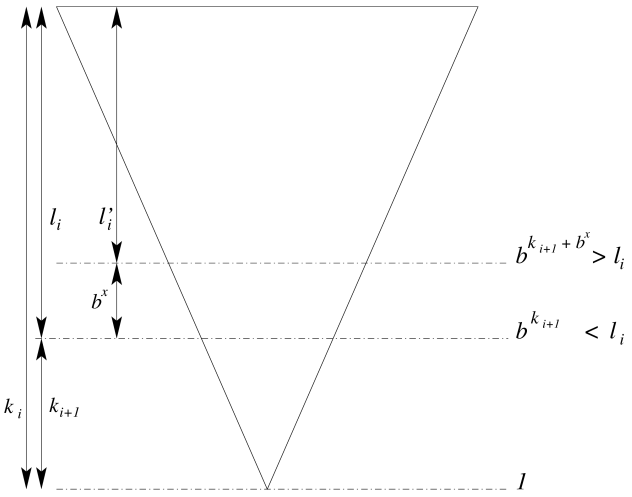

Figure 4. Step i of the construction of mapping μ_1.

integral multiple of l_i because l_i is also a power of b. Thus, we can map the trees of the last l'_i levels evenly onto the l_i processors. For the remaining b^x levels to be mapped in this step, we map those levels starting with the level closest to the root having $b^{k_{i+1}}$ nodes: we map each node onto one processor, using $b^{k_{i+1}}$ processors. For the next level having $b \cdot b^{k_{i+1}}$ nodes, we map b nodes on each processor, using another $b^{k_{i+1}}$ processors. When we have finished with those b^x levels, we have used $b^x \cdot b^{k_{i+1}} = l_i$ processors. Note that this straightforward placement corresponds to applying mapping μ_0 for the $b^{k_{i+1}}$ trees of $k = b^x$ levels each, with the processor capacity scaled down to $b^{-k_{i+1}}$. We might also apply mapping μ_1 recursively to further balance the load.

On each processor, we have placed a load of $l'_i/l_i = 1 - b^{-k_{i+1}}$ by mapping l'_i levels, and $b^{-k_{i+1}}$ by mapping the first b^x levels. It follows that the computational load on each processor is 1. The maximum memory load is determined in step $i = 0$, because the majority of the nodes is mapped there. In this step $(b^k - b^{k-l_0})/(b-1)$ nodes are mapped onto l_0 processors, so that each processor receives a memory load of

$$\frac{b^k - b^{k-l_0}}{(b-1)l_0} < b \cdot \frac{b^k - 1}{(b-1)k}$$

because $l_0 \geq k/b$. Thus, the memory load is larger than the lower bound by a factor less than b. Note that this is not completely exact because the b^x levels — if they are used in the first step — are not mapped with a completely even memory load. However, the imbalance is only very slight, as our simulations will show.

In each step, there are at most two levels (the first one

of the b^x and the first one of the l'_i) where siblings are not placed on the same processor.

As in each step i the largest power of b less than k_i is chosen as the number l_i of levels mapped, the number r of steps made by the mapping algorithm is one plus the cross sum of $k-1$ in b-ary representation, and thus $r \leq 1+(b-1)\cdot \log_b(k-1)$.

We summarize the properties of mapping μ_1:

Lemma 2 *The maximum computational load of mapping μ_1 is $C^*_{\mu_1} = 1$, which is optimal.*

The maximum memory load of mapping μ_1 is about $M^*_{\mu_1} = \frac{b^k - b^{k-l_0}}{(b-1)l_0}$, which is larger than the lower bound by a factor of less than b.

In at least $k-2r$ levels, siblings are mapped to the same processor, where $r \leq 1+(b-1)\log_b(k-1)$ is the number of the steps in the construction of the mapping.

We illustrate the mapping algorithm for the case $b=2$ and $k=5$. The resulting mapping is identical to the one in Fig. 2, i.e. the approximation algorithm produces an optimal result. In step $i=0$, we have $l_0=4$ as this is the largest power of 2 less than $k_0 = k = 5$. Hence, $k_1 = 1$. The levels to be mapped consist of $2^{k_1}=2$ trees of 4 levels, and thus cannot be mapped directly. It follows that $x=1$ as $l_0 = 4 = 2^1 \cdot 2^1 = 2^{k_1}\cdot 2^x$, and thus $l'_0 = l_0 - 2^x = 2$. The last two levels of the 5-level tree consist of 8 trees, so that two of them are mapped onto each processor. Then we place the remaining $2^x = 2$ levels, of which the first consists of two nodes, that are mapped onto two processors, one node on each processor. The last level consists of 4 nodes, of which 2 are mapped on each of two processors. Finally, in step $i=1$, we have $k_1 = 1$ and map the root onto the last processor. The maximum memory load of the mapping is 8 which is optimal (see previous section) although it is a factor of 1.29 away from the lower bound.

As a second example we consider $k=8$ and $b=2$. In this example $b^{k_{i+1}} \geq l_i$ for all steps i. In step 0, we map $l_0 = 4$ levels of the tree onto 4 processors, in step 1 we map $l_1 = 2$ levels, and in steps 2 and 3 we map 1 level, respectively. The resulting mapping is depicted in Fig. 5. The maximum computational load on each processor is 1, which is optimal, and the maximum memory load is 60, on processors 0 to 3, which is a factor of 1.9 away from the lower bound.

Both examples were chosen in part because they represent two extremes: $k = 5 = 2^2+1$ is a power of two plus one, and thus l_0 can be chosen the maximum value so that $k_1 = 1$, and the mapping can be constructed in two steps. The closer l_0 is to k, also the closer the maximum memory load is to the lower bound. In contrast, for $k=8$, we must choose $l_0 = 4$ which is only half of k, and thus the worst value possible. As a consequence the maximum memory load is by a factor of 1.88 larger than the lower bound.

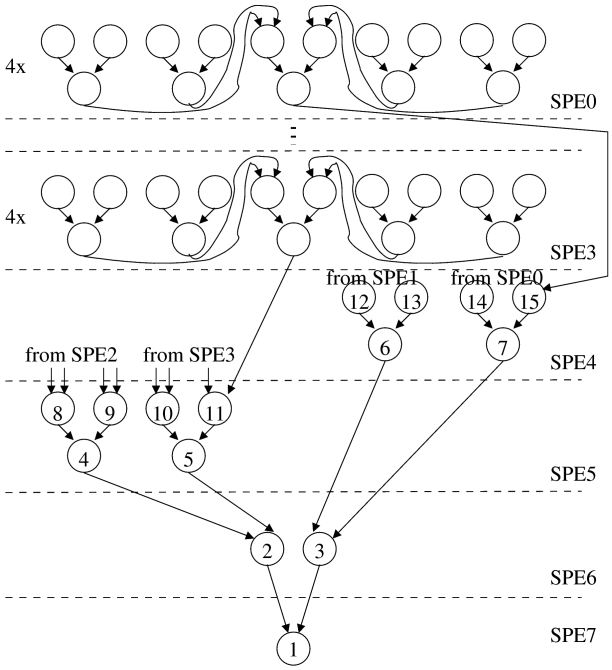

Figure 5. Mapping a 8-level binary tree onto 8 processors.

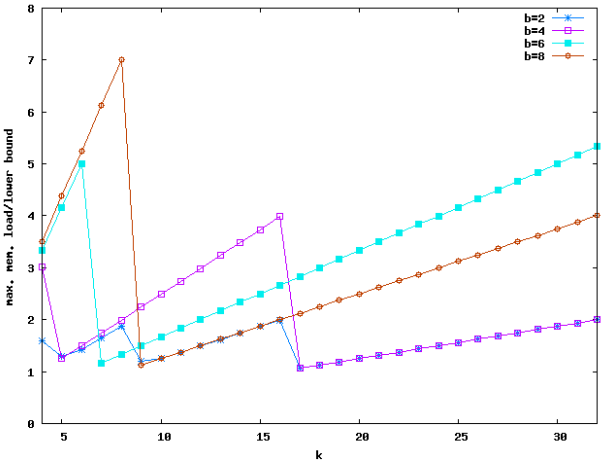

Figure 6. Ratio between max. memory load and lower bound depending on k and b.

In general, the ratio between maximum memory load and lower bound increases with increasing k in intervals $[b^j+1,\ldots,b^{j+1}]$ from 1 to b. We have illustrated this for $k = 4,\ldots,32$ and $b = 2,4,6,8$ in Fig. 6.

Several cases remain to be considered. In the case that $p < k$, there are several possibilities. If k is a multiple of p, then we could first construct a mapping onto k pseudo-

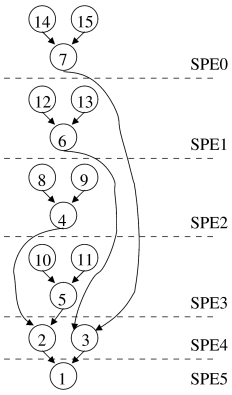

Figure 7. Mapping a 4-level tree onto 6 SPEs.

processors, and then map k/p of those pseudo-processors onto one processor. Alternatively, one could split the k-level tree into p-level sub-trees that can be mapped as before, and have the system work in a scheduled way, e.g. round-robin, on the sub-trees. If k is not a multiple of p, then we again split the tree into p-level sub-trees starting from the leaves. The sub-tree containing the root will contain only $k \bmod p < p$ levels, a case which is treated below.

Note that k is an input parameter of our problem as the sorting algorithm in principle is free to choose k as seems suitable. Therefore k should be chosen such it fits well with the available number of processors, i.e. in most cases one will try to choose $k = p$.

In the case that $p > k$, there are again several possibilities. If p is a multiple of k, then one could first construct a mapping onto k pseudo-processors, and then distribute the work of each pseudo-processor evenly onto p/k processors. Note that distributing the work of the pseudo-processor to which the root node is mapped consists of parallelizing a single merge node. However, there are algorithms known in the literature for that problem, see e.g. Chapter 4.2 of [7] that presents a parallel merge on n processors with $O(n \log n)$ work.

In case that p is only slightly larger than k, one may also think of distributing the work of processors to which the leaves are mapped, onto several processors. The reduction of the computational load for these processors takes into account that the computational load is an *average* over the time, and can compensate variations too large to be taken care of by buffering. An example is depicted in Fig. 7 for $k = 4$ and $p = 6$. The mapping for $k = p = 4$ uses $l_0 = 2$ in the first step, but distributes the load for two processors now onto four processors.

So far, we have only considered the memory load, and not the load on the ring network. The processor P_{p-1} holding solely the root node will have an output rate of 1, which is transported over the ring to the external memory. Each set of processors maps a number of levels. The rate of the communication leaving those processors sums up to 1 as well, as this is the load on each level. Hence, no part of the ring will have a higher network load, so that also the network load scales well with the algorithm.

5. Related Work

Partitioning and mapping of task graphs is, in general, a NP-complete problem and has been discussed a lot in the literature.

One application area is, as in our case, the parallelization of programs with given dependence graph for execution on a (mostly, shared memory) parallel computer, with the objective to balance the work load of the partitions, minimize the number of partitions (aka. *processor minimization*), and/or minimize the overall weight of all edges cut by the partitioning, as all these are supposed to correspond to expensive shared memory accesses (aka. *bandwidth minimization*).

Another related area is the (spatial) clustering of logic circuits into partitions each matching a maximum chip size constraint, while the communication between partitions must fit an upper limit on the number of pins per chip. Here, one is (as in our case) mainly interested in reducing the accumulated weight of all edges cut between any two adjacent partitions (aka. *bottleneck minimization*).

There is a wealth of literature on mapping and scheduling acyclic task graphs of streaming computations to multiprocessors. Some methods are designed for special topologies, such as linear chains and trees, while others address general task graphs.

Mapping of special topologies For *tree-shaped task graphs*, various partitioning algorithms have been proposed.

Bokhari [8] considers partitioning of trees for master-slave (there called host-satellite) systems where the partition containing the root is mapped to the master (host) processor while the slaves (satellites) are each assigned exactly one complete subtree that is connected directly to the master partition.

Ray and Jiang [9] show that the bandwidth minimization problem is NP-complete even for trees, and give a fast heuristic algorithm for it. In a follow-up paper [10], the same authors give polynomial-time greedy algorithms for bottleneck minimization and processor minimization of tree task graphs.

Most approaches for tree partitioning are for non-pipelined trees and therefore assume that the tree partitions should be connected components (i.e., contiguous subtrees) and exactly one partition be mapped to one processor. This does not apply in our case, where partitions can consist of multiple disconnected subtrees, so that processors could be better "filled up" to their computational capacity with residual tree fragments if this improves system throughput. Also,

in our scenario the b-ary tree is always complete, thus we can exploit symmetry properties that are not given in the more general case.

Lüling et al. [11] consider the problem of mapping a tree that evolves in a search problem onto a distributed memory parallel computer in such a way that computation and communication times both are minimized. They focus on trees that evolve dynamically, i.e. are not known beforehand as in our case. The work associated with each tree node seems to be constant while the computational load in our case depends on the tree level of the node. As the tree is not kept completely, memory load plays a minor role. In contrast, we map a tree to be kept completely in memory. Finally, the trees considered in search problems typically are far from balanced and their degree is irregular, while we consider balanced b-ary trees.

Middendorf et al. [12] consider non-pipelined, tree-like task graph structures such as reduction trees, task graphs for parallel prefix computations and *Butterfly graphs*, under the LogP cost model that accounts for transfer latency and limited communication bandwidth in message passing systems. They give polynomial-time algorithms for computing optimal schedules for special cases. However, memory constraints or pipelined versions of these task graphs are not considered.

Mapping of general task graphs The approaches for mapping general task graphs can be roughly divided into two classes: Non-overlapping scheduling and overlapping scheduling.

Non-overlapping scheduling schedules a single execution of the program (and repeats this for further input sets if necessary); it aims at minimizing the makespan (execution time for one input set) of the schedule, which depends strongly on task and communication latencies, while memory constraints are usually a non-issue here. A typical result is that all tasks on a critical path are mapped to the same processing unit. The mapping and scheduling can thus be done by classical list-scheduling based approaches for task graph clustering that attempt to minimize the critical path length for a given number of processors. Usually, partitions are contiguous subgraphs. The problem complexity can be reduced heuristically by a task merging pre-pass that coarsens the task granularity. Optimization methods applied include e.g. gradient search as in Sarkar and Hennessy [13]. See [14] for a recent survey and comparison.

Szymanek and Kuchcinski [15] propose a heuristic method for memory-aware assignment and scheduling of a task graph to a bus- or link-connected set of processing units. Tasks are parametrized in their code and data memory needs, and edges between tasks by the buffer space requirements on sender and receiver side during the whole communication period that results if an edge is selected as communication edge between partitions. Based on initial estimations for maximum data memory use, this iterative optimization method toggles between two strategies for assignment and scheduling, namely critical path scheduling (which optimizes for the makespan) and scheduling for minimization of memory usage, trying to balance execution time and memory utilization of the resulting solution.

Overlapping scheduling, which is closely related to *software pipelining* [16, 17] and *systolic parallel algorithms* [18], instead overlaps executions for different input sets in time and attempts to maximize the throughput in the steady state, even if the makespan for a single input set may be long. Mapping methods for such pipelined task graphs, especially for signal processing applications in the embedded systems domain, have been described e.g. by Hoang and Rabaey [19] and Ruggiero *et al.* [20]. Our method also belongs to this second category.

Hoang and Rabaey [19] work on a hierarchical task graph such that task granularity can be refined by expanding function calls or loops into subtasks as appropriate. They provide a heuristic algorithm based on greedy list scheduling for simultaneous pipelining, parallel execution and retiming to maximize throughput. The resulting mapped pipeline is a linear graph where each pipeline stage is assigned one or several processors. Buffer memory requirements are considered only when checking feasibility of a solution, but are not really minimized for. The method only allows contiguous subDAGs to be mapped to a processor.

Ruggiero *et al.* [20] decompose the problem into mapping (resource allocation) and scheduling. The mapping problem, which is close to ours, is solved by an integer linear programming formulation, too, and is thus, in general, not constrained to partitions consisting of contiguous subDAGs as in most other methods. Their framework targets MPSoC platforms where the mapped partitions form linear pipelines. Their objective function for mapping optimization is minimizing the communication cost for forwarding intermediate results on the internal bus. Buffer memory requirements are not considered.

6. Conclusion

We have shown how to lower memory bandwidth requirements in code for the Cell BE by on-chip pipelining of memory-intensive computations. To realize pipelining with maximum throughput while reducing on-chip memory load and interprocessor communication, we formulated a general optimization problem for mapping task graphs. We have demonstrated our model with case studies from data-parallel code generation and merge trees in sorting. Small to medium sized problem instances can be solved optimally by ILP, larger ones by heuristics and approximation algorithms. We have also presented a new tree-specific approx-

imation algorithm for the mapping problem.

Implementing and evaluating the resulting code on Cell is an issue of current and future work. The method could be used e.g. as an optimization in code generation for dataparallel code in an optimizing compiler for Cell, such as [21].

Acknowledgements C. Kessler acknowledges partial funding by Vetenskapsrådet, SSF, Vinnova, and CUGS.

References

[1] Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation—a performance view. IBM J. Res. Devel. **51**(5) (Sept. 2007) 559–572

[2] Kessler, C.W., Keller, J.: Optimized mapping of pipelined task graphs on the Cell BE. In: Proc. 14th Int. Workshop on Compilers for Parallel Computing (CPC-2009), Zürich, Switzerland. (January 2009)

[3] Keller, J., Kessler, C.W.: Optimized pipelined parallel merge sort on the Cell BE. In: Proc. 2nd Workshop on Highly Parallel Processing on a Chip (HPPC-2008) at Euro-Par 2008, Gran Canaria, Spain. (2008)

[4] Gedik, B., Bordawekar, R., Yu, P.S.: Cellsort: High performance sorting on the Cell processor. In: Proc. 33rd Int.l Conf. on Very Large Data Bases. (2007) 1286–1207

[5] Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In: Proc. 16th Int.l Conf. on Parallel Architecture and Compilation Techniques (PACT), IEEE Computer Society (2007) 189–198

[6] ILOG Inc.: CPLEX v. 10.2. www.ilog.com (2007)

[7] JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley (1992)

[8] Bokhari, S.H.: Partitioning problems in parallel, pipelined and distributed computing. IEEE Transactions on Computers **37**(1) (January 1988)

[9] Ray, S., Jiang, I.: Improved algorithms for partitioning tree and linear task graphs on shared memory architecture. In: Proceedings of the 14th International Conference on Distributed Computing Systems. (June 1994) 363–370

[10] Ray, S., Jiang, I.: Sequential and parallel algorithms for partitioning tree task graphs on shared memory architecture. In: Proc. International Conference on Parallel Processing, Volume 3. (August 1994) 266–269

[11] Lüling, R., Monien, B., Reinefeld, A., Tschöke, S.: Mapping tree-structured combinatorial optimization problems onto parallel computers. In: Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques, London, UK, Springer-Verlag (1996) 115–144

[12] Middendorf, M., Löwe, W., Zimmermann, W.: Scheduling inverse trees under the communication model of the LogP-machine. Theoretical Computer Science **215** (1999) 137–168

[13] Sarkar, V., Hennessy, J.: Compile-time Partitioning and Scheduling of Parallel Programs. In: Proc. ACM SIGPLAN Symp. on Compiler Construction. (1986) 17–26

[14] Kianzad, V., Bhattacharyya, S.S.: Efficient techniques for clustering and scheduling onto embedded multiprocessors. IEEE Trans. on Par. and Distr. Syst. **17**(7) (July 2006) 667–680

[15] Szymanek, R., Kuchcinski, K.: A constructive algorithm for memory-aware task assignment and scheduling. In: CODES '01: Proc. 9th int. symposium on Hardware/software codesign, New York, NY, USA, ACM (2001) 147–152

[16] Rau, B., Glaeser, C.: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: Proc. 14th Annual Workshop on Microprogramming. (1981) 183–198

[17] Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. In: Proc. ACM SIGPLAN Symp. on Compiler Construction. (July 1988) 318–328

[18] Kung, H.T.: Why systolic architectures? IEEE Computer **15** (January 1982) 37–46

[19] Hoang, P.D., Rabaey, J.M.: Scheduling of DSP programs onto multiprocessors for maximum throughput. IEEE Trans. on Signal Processing **41**(6) (June 1993) 2225–2235

[20] Ruggiero, M., Guerri, A., Bertozzi, D., Milano, M., Benini, L.: A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness. Int. J. of Parallel Programming **36**(1) (February 2008)

[21] Eichenberger et al., A.E.: Using advanced compiler technology to exploit the performance of the Cell Broadband Engine (TM) architecture. IBM Systems Journal **45**(1) (2006)