Optimized Mapping of Pipelined Task Graphs on the Cell BE

Christoph W. Kessler[1] and Jörg Keller[2]

[1] Linköpings Universitet, Dept. of Computer and Inf. Science, 58183 Linköping, Sweden
[2] FernUniversität in Hagen, Dept. of Math. and Computer Science, 58084 Hagen, Germany

Abstract. Limited bandwidth to off-chip main memory poses a problem in chip multiprocessors for streaming applications, such as Cell BE, and will become more severe with the expected increase in the number of cores. Especially for streaming computations where the ratio between computational work and memory transfer is low, the generation of memory-efficient code is thus an important compiler optimization.

We suggest to use pipelining between the SPEs over the high-bandwidth internal bus of Cell BE to reduce the required main memory bandwidth, and thereby improve the computation throughput for memory-intensive computations. At the same time, we are constrained by the limited size of SPE on-chip memory available for additional buffers that are necessary for the pipelining between SPEs.

We investigate mappings of the nodes of a pipelined parallel task graph to the SPEs that are optimal trade-offs between load balancing, buffer memory consumption, and communication load on the on-chip bus. We solve this multi-objective optimization problem by deriving an integer linear programming (ILP) formulation and compute Pareto-optimal solutions for the mapping with a state-of-the-art ILP solver. For larger problem instances, we sketch a two-step approach to reduce problem size.

We exemplify our mapping technique with several memory-intensive example problems: with acyclic pipelined task graphs derived from data parallel code, with complete d-ary tree pipelines for parallel mergesort on Cell BE, and with butterfly pipelines for parallel FFT on Cell BE. We validate the mappings with discrete event simulations.

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. The bandwidth to the external memory is much smaller than the SPEs' aggregate bandwidth to the on-chip interconnect bus (EIB) [1]. This limits performance and prevents scalability. External memory is also a bottleneck in other multiprocessors-on-chip. This problem will become more severe as the core count per chip is expected to increase considerably

in the foreseeable future. Scalable parallelization on such architectures therefore must use communication between the SPEs to reduce communication with external memory.

In this paper, we consider the important domain of streaming computations such as sets of dataparallel computations on large vectors, pipelined mergesort, and pipelined FFT computations. In contrast to the textbook-style parallel formulations of these algorithms as a sequence of operations on complete arrays, these computations should be reorganized in a pipelined fashion, such that intermediate results (temporary vectors) are not written back to main memory but instead forwarded immediately to a consuming successor operation. This will require some buffering of intermediate results in on-chip memory, but this is necessary anyway in processors like Cell in order to overlap computation with (DMA) communication. It also requires that all tasks (elementary streaming operations) of the algorithm be active simultaneously; however, as we would like to guarantee fast context switching on SPEs, the limited size of Cell's local on-chip memory then puts a limit on the number of tasks that can be mapped to an 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. Hence, we obtain a constrained optimization problem for mapping the tasks of streaming computations to the SPEs of Cell such that the resulting throughput is maximized.

For a specific algorithm such as parallel mergesort, this transformation may require some human guidance e.g. to identify the individual tasks with their properties, the multi-way buffering scheme to be used etc. [2]. For other cases such as dataparallel computations, such information may be derived automatically by a compiler. As we have shown in previous work on pipelined parallel mergesort, a good or even optimal placement of the tasks of the resulting tree-shaped pipelined computation can be computed by approximation algorithms or integer linear programming [2,3]. In this paper, we generalize this approach to the automatic computation of optimal or near-optimal placements of tasks of arbitrary acyclic streaming task graphs. Such general graphs may arise e.g. from the array data flow between dataparallel operations that have been identified by the programmer or by a vectorizing compiler. We present a solution for the mapping problem based on integer linear programming and demonstrate the method for examples from dataparallel computations, stream-based sorting, and FFT computations.

The remainder of this article is organized as follows. In Section 2, we will characterize some example classes from the streaming computation domain that differ in graph topology and data rates, and that will be used throughout the paper to demonstrate our mapping technique. In Section 3, we formalize the general mapping problem, give an integer linear programming based solution to the mapping problem, and report on first experimental results. We sketch a heuristic to partition larger problem instances. Section 4 reviews related work, and Section 5 concludes.

2 Streaming Computations: Three Case Studies

2.1 Dataparallel Computations

A dataparallel operation applies the same scalar operation to each element of an input sequence (or of several input sequences, in case of operations of higher arity) of some

length N to produce an output sequence of length N. All element computations are independent and take the same (short) time. Dataparallel operations can either be derived from inner loops of a sequential program by a vectorizing compiler, or explicitly identified by the programmer in the form of vector instructions in array syntax, forall loops, or special intrinsic functions, in a dataparallel programming language. Dataparallel operations are especially suitable for Cell because they generally allow to use SIMD instructions on the SPEs, running up to 4 floatingpoint operations per clock cycle on adjacent data.

Subsequent dataparallel operations may consume a part or (most typically) the entire result produced by a predecessor operation. These dataparallel operations are thus coupled by array data flow, usually in the form of (temporary) array variables. For an acyclic sequence of dataparallel operations in a program, the dataparallel operations with the (array) data flow edges between them define an acyclic task graph. Here, we focus on the *quasiscalar* case [4] where the entire dataparallel task graph could be considered a single dataparallel operation, applying a complex elementwise computation to every input element of all input arrays. In other words, the dataparallel task graph can be seen as an overlay of N independent scalar task graphs; in particular, it can be arbitrarily partitioned along the length dimension. All data rates along array data flow edges are equal, as N elements are forwarded along each edge. The (relative) computational load that a dataparallel task in such a dataparallel task graph represents is determined only by the cost of the elementary operation applied per input element. In many cases, these can be considered equal, too. (Special treatment can be done for reduction operations that consume an input sequence of length N but produce just a single output element by accumulating over the input, such as vector sums. For these, we simply neglect the size of the output stream, as it is small compared to N.)

Computing a sequence of dataparallel operations in the traditional (vector-style) way will process each dataparallel operation completely before proceeding with the next one. Independent dataparallel operations (i.e., whose nodes are not located on the same path in the task graph) could be executed simultaneously by different processors (SPEs). For instance, the dataparallel task graph could be scheduled level-wise. Alternatively, each dataparallel operation could be cut into p partitions, each computed by one SPE in parallel. Either way, all intermediate results will then be stored in off-chip memory, requiring a higher overall communication volume, such that the off-chip memory bandwidth limits the number of dataparallel computations that can run simultaneously on the Cell BE at maximum speed.

In contrast, pipelining a dataparallel computation means that all tasks will be active at the same time, each requiring a certain CPU (SPE) share that is proportional to its computational load in the steady state of the pipeline computation where all tasks have input data to work on. A fair round-robin user-level scheduler on each SPE guarantees that, in the steady state, each of multiple tasks mapped to the SPE will use its assigned share of processing time. The pipelining also implies that small packets of intermediate results need to be buffered in on-chip memory to account for small jitter in the processing speed of each task. If the buffers are sufficiently large, no task will be slowed down due to waiting for input packets or for flushing output packets, as the average processing rates are balanced over all tasks. The advantage of pipelining is that the communication

to off-chip memory is minimized. Its downside is a possibly high demand of on-chip memory space for buffering packets with intermediate results.

To allow for overlapping DMA handling of packet forwarding (both off chip and on chip) with computation on Cell, there should be at least buffer space for 2 input packets per input parameter and 2 output packets per output parameter of each dataparallel operation. This amounts to at least 6 packet buffers for an ordinary binary dataparallel operation. On Cell, the DMA packet size cannot be made arbitrarily small: the absolute minimum is 16 bytes, and in order to be not too inefficient, at least 128 bytes should be shipped at a time. Reasonable packet sizes are a few KB in size (the upper limit is 16KB). As the size of SPE local storage is severely limited (256KB for both code and data) and the packet size is the same for all SPEs and throughout the computation, this means that the maximum number of packet buffers of the tasks assigned to any SPE should be as small as possible. On the other hand, excessively large packet sizes mean a higher pipeline startup time. However, as long as N is very large compared to the length of the critical path in the task graph, the startup time penalty can be neglected.

2.2 Stream-based Sorting

Sorting is an important subroutine in many high performance computing applications, and parallel sorting algorithms on a wealth of architectures have therefore attracted considerable interest continuously for the last decades, see e.g. [5,6]. As the ratio between computation and transfer to memory is quite low in sorting, it presents an interesting case study to develop bandwidth efficient algorithms. Sorting algorithms implemented for the Cell BE [7,8] work in two phases to sort a data set of size N with local memories of size N'. In the first phase, blocks of data of size $8N'$ that fit into the combined local memories of the 8 SPEs are sorted. In the second phase, those sorted blocks of data are combined to a fully sorted data set. We concentrate on the second phase as the majority of memory accesses occurs there. In [7], this phase is realized by a bitonic sort because this avoids data dependent control flow and thus fully exploits SPE's SIMD architecture. Yet, $O(N \log^2 N)$ memory accesses are needed and the reported speedups are small. In [8], mergesort with 4-to-1-mergers is used in the second phase. The data flow graph of the merge procedures thus forms a fully balanced merge quadtree. As each merge procedure on each SPE reads from main memory and writes to main memory, all N words are read from and written to main memory in each merge round, resulting in $N \log_4(N/(8N')) = O(N \log_4 N)$ data being read from and written to main memory. While this improves the situation, speedup still is limited.

In order to overcome this bottleneck, we run merger nodes of consecutive layers of the merge tree concurrently, so that output from one merger is not written to main memory but sent to the SPE running the follow-up merger node, i.e. we use pipelining. If we can embed subtrees with k levels in this way, we are able to realize parallelized 4^k-to-1 merge routines and thus increase the ratio of computation to memory transfer by a factor of k. Yet, this must be done in a manner such that all SPEs are kept busy, and that the overheads introduced are not too high. Our approach is that a merger node does not process complete blocks of data before forwarding its result block, but uses fixed sized chunks of the blocks, i.e. a merger node is able to start work as soon as it has one chunk of each of its input blocks, and as soon as it has produced one chunk of the

output block, it forwards it to the follow-up node. This form of streaming allows to use fixed size buffers, holding one chunk each. In order to be able to overlap data transfer and computation, the merger nodes should use double buffering at least for their inputs. Also, the chunks and thus the buffers should have a reasonable minimum size to allow for efficent data transfer between SPEs. Note that due to the small local memory on SPEs, already the mergers in the algorithm of [8] must work with buffering and chunks.

Furthermore, ensuring that our pipeline runs close to the maximum possible speed requires consideration of load balancing. If a b-ary merger node u must provide an output rate of τ words per time unit, then its predecessor mergers $u_1, ..., u_b$ feeding its inputs must provide a rate of τ/b on average. However, if the values in the output chunk produced by u_i are much larger than those in u_j, the merger u will only process values from the output chunk of u_j for some time, so that u_j must produce at a double rate for some time, while u_i will be stalled because of finite buffering between u_i and u. Otherwise the rate of u will reduce.

Finally, the merger nodes should be distributed over the SPEs such that not all communication between merger nodes leads to communication between SPEs, in order not to overload the EIB.

2.3 FFT Computations

Fast Fourier Transform (FFT) is a divide-and-conquer based algorithm to compute the Discrete Fourier Transform (DFT) (see e.g. [9,10] for descriptions and parallelization). The task graph of Fast Fourier Transform (FFT) is recursively defined as follows: An elementary FFT butterfly computation takes 2 complex elements as input, performs 3 arithmetic operations (multiplication with a tabled twiddle factor, followed by an addition resp. subtraction) and produces 2 complex elements as output. An even number $q > 1$ of input elements to be processed by a size-q-FFT is split in subvectors of $q/2$ odd-indexed and $q/2$ even-indexed elements. These subvectors can be processed independently and recursively by $(q/2)$-FFT subcomputations if $q/2$ is even, otherwise a DFT may be used instead. Each subcomputation produces $q/2$ output elements that are elementwise combined with butterfly computations to produce the $q/2$ lower-indexed and $q/2$ upper-indexed result elements of the q-FFT. Hence, unrolling the recursion for q being a power of 2, the task graph becomes a butterfly graph with $(q/2)\log_2 q$ butterfly tasks, where all node weights are equal and all edge weigths are equal. In contrast to mergesort, no load imbalances can occur here, as FFT is an oblivious algorithm.

The mismatch of input to output indices can be fixed by a bit reversal operation (swapping every input element at position i with the one at position $\bar{i}$ whose binary representation is just the mirrored one of i) before starting the actual (unordered FFT) computation, which now can produce contiguous blocks of data and is easily expressed in an iterative formulation. If all contiguous butterfly computations residing on the same level of the corresponding task graph and expressible by a single dataparallel operation are merged, the resulting task graph for this iterative formulation has a tree topology where data rates along edges and node weights look the same way as for the merge trees described above; hence we focus on the recursive FFT butterfly task graphs in the remainder of this work.

3 Mapping Pipelined Task Graphs onto Processors

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 three classes of streaming computations described earlier. We also give a heuristic mapping algorithm for larger task graphs and machine configurations.

3.1 Definitions

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 merge-sort 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 is basically proportional to the number of DMA packet buffers that it requires; for simplicity, we abstract from the size of other data structures of the tasks and their code, for which we reserve a fixed, small share of each SPE local store. We distinguish between two scenarios: (1) The *mapping-invariant memory load model* assumes a fixed memory load value depending on the computation type of v; specifically, 4 buffers for unary nodes, 6 for binary nodes, and 0 for loads and stores, representing the double buffering policy to be applied for operand and result streams. This model is somewhat conservative but leads to a less complex optimization problem instance. (2) The *mapping-sensitive memory load model* accounts for the fact that buffering output data

is only needed for tasks with at least one successor placed on another SPE, while for the others, output can be written directly into the free input buffer of the successor task in the same SPE local store (where the DMA transfer as well as the writing task may have to wait in favor of e.g. its successor when no input buffer is free). Here, we assign a base memory load $\beta_b(v)$ of 2 for unary and 4 for binary operations, 0 for loads, and 2 for stores to account for their predecessor's output buffering to main memory. To this, we add a binary mapping-specific load component $\beta_\mu(v)$ that is 1 iff at least one successor of task v is placed on another SPE. Hence, $\beta(v) = \beta_b(v) + \beta_\mu(v)$. This also implies that shared loads have memory load 1.

In homogeneous task graphs such as merge trees or FFT butterflies, all $\beta(v)$ are equal under the mapping-invariant memory load model. 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.

3.2 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.

Our ILP formulation for the mapping-invariant memory load model uses two arrays of boolean variables, x and z. For the mapping-sensitive memory load model, we add a third array, y.

The actual solution, i.e. the mapping μ of nodes to processors, will be given by x: $x_{v,q} = 1$ iff node v is mapped on processor q.

In order to determine which edges are internal (i.e., where both source and target node are mapped to the same processor), we introduce auxiliary variables z and y:
$z_{(u,v),q} = 1$ iff both source u and target v of edge (u, v) are mapped to processor q.
$y_{u,q} = 1$ iff u is mapped to q and some successor of u is mapped to some other SPE.

Also, we use an integer variable *maxMemoryLoad* that will indicate the maximum memory load assigned to any SPE in P, and a linear variable *maxComputLoad* that indicates the maximum accumulated load mapped to a processor.

The following constraints must hold:

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 (here for the mapping-sensitive model) should be balanced:

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

Communication cost occurs whenever an edge is not internal, i.e. its endpoints are mapped to different SPEs. To avoid products of two x variables when determining which edges are internal, we use the following $2mp$ constraints:

$$\forall (u, v) \in E,\ q \in P: \quad z_{(u,v),q} \leq x_{u,q} \quad \text{and} \quad z_{(u,v),q} \leq x_{v,q}$$

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 anyway.

The communication load is the total communication volume over all edges minus the volume over the internal edges:

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

We apply the same construction to determine $y_{v,q}$:

$$\forall (u, v) \in E,\ q \in P: \quad y_{u,q} \geq x_{u,q} - x_{v,q} \ \text{ and } \ y_{u,q} \geq x_{v,q} - x_{u,q}$$

Finally, the objective function is:

$$\text{Minimize} \quad \Lambda \cdot \textit{maxComputLoad} + \epsilon_M \cdot \textit{maxMemoryLoad} + \epsilon_C \cdot \textit{commLoad} + M$$

where Λ is a value large enough to prioritize computational load balancing over all other optimization goals, and the positive weight parameters $\epsilon_M < 1$ and $\epsilon_C < 1$

can be set appropriately to give preference to minimizing for *maxMemoryLoad* or for *commLoad* as secondary optimization goal. The formulation requires that $\epsilon_C > 0$. For the mapping-sensitive memory load model, we add the term $M = \sum_{u \in V,\ q \in P} y_{u,q}$.

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:

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

In total, the formulation for the mapping-invariant memory load model uses $np + mp = O(np)$ boolean variables, 1 integer variable, 2 linear variables, and $2mp + 2p + 2 = O(np)$ constraints. For the mapping-sensitive memory load model, it has $2np+mp$ boolean variables and $4mp + 2p + 2$ constraints.

We implemented the above ILP model in CPLEX 10.2 [11], a commercial ILP solver. In the next subsections, we will report on the results obtained.

3.3 ILP Optimization Results for Merge Trees

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 [2]. Table 1 shows all Pareto-optimal solutions that CPLEX found for $k = p = 5, 6, 7$, using the mapping-invariant memory load model. While the majority of computations 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 1 shows two of the solutions for $k = 5$, visualized with a tool developed by us.

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

k	5			6				7		
# binary var.s	305			750				1771		
# constraints	341			826				1906		
maxMemoryLoad	8	9	10	13	14	15	20	21	29	30
commLoad	2.5	2.375	1.75	2.625	2.4375	1.9375	1.875	2.375	2.3125	2.0

To test the performance of our 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 used scheduling and buffering as described in Subsection 2.1 with buffers holding 1,024 integers of 32 bits. We have investigated several mappings resulting from our mapping algorithm. The 5-level tree of Fig. 1(b) realizes a 32-to-1 merge. This seems the limit with a memory load of 8 nodes, and 16 to 24 KB of buffering per merger node (4 or 6 buffers of 4 KB each). With input blocks of 2^{20} randomly chosen,

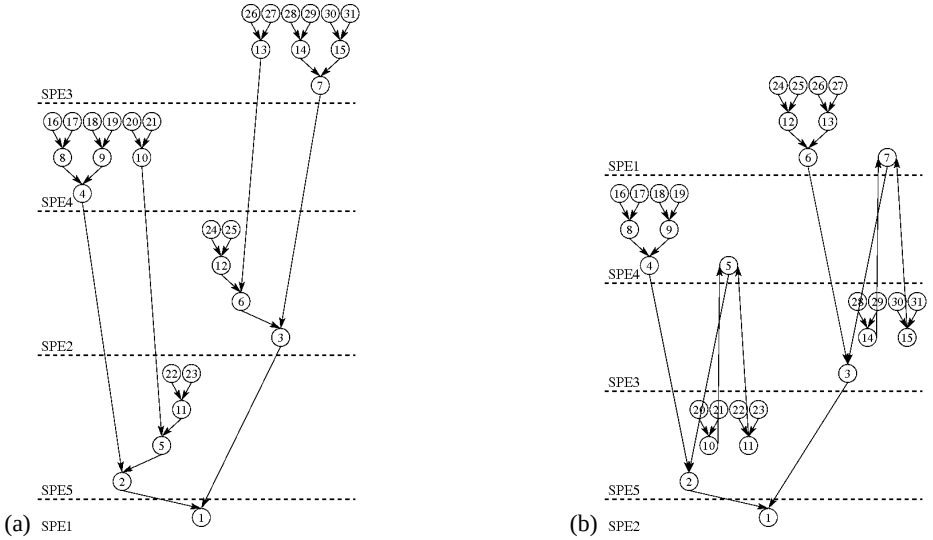

Fig. 1. Two Pareto-optimal solutions for mapping a 5-level merge tree onto 5 processors, computed by the ILP solver. (a) max. memory load 10 and communication load 1.75, obtained e.g. for $\epsilon_M = 0.1\epsilon_C$; (b) max. memory load 8 and communication load 2.5, obtained e.g. for $\epsilon_M = 10\epsilon_C$.

sorted integers, the pipeline efficiency was 93%, and thus lowered bandwidth requirements to main memory by a factor of 1.86 compared to [8]. For further results for mapped merge trees, see [2].

3.4 ILP Optimization Results for FFT Butterfly Graphs

Figure 2 shows a Pareto-optimal mapping computed for a 8×4 nodes Butterfly graph onto 6 processors with the mapping-invariant memory load model. For the visualization of Butterfly graph mappings, we have written a converter to `dot` format and use the `graphviz` package to derive a drawing where boxes enclose nodes mapped to the same SPE. Normalized node weights are given within node circles, and normalized edge weights are attached to the edges. We simulated this mapping in a discrete event simulation and achieved a pipeline efficiency of close to 100%.

3.5 ILP Optimization Results for Dataparallel Computations

In order to test the ILP model with dataparallel task graphs, we used several hand-vectorized fragments from the Livermore Loops [12] and synthetic kernels, see Table 2. We focused on the easily vectorizable kernels and applied only standard vectorization methods, hence any reasonable vectorizing compiler could have produced the same dataparallel task graph. Such graphs are usually of very moderate size, and computing an optimal ILP solution for a small number of SPEs thus takes just a few seconds in most of the cases, at least with the mapping-invariant memory load model. For two

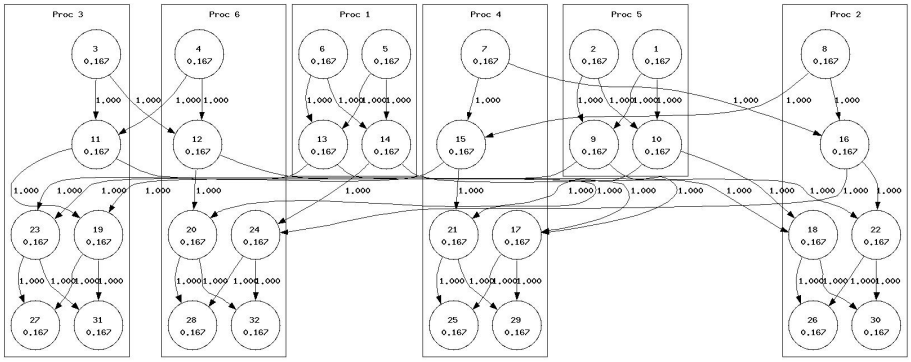

Fig. 2. A Pareto-optimal solution for mapping a 8×4-Butterfly graph onto 6 processors, computed from the ILP model with 482 variables and 621 constraints in 4 seconds.

Table 2. ILP models for dataparallel task graphs extracted from the Livermore Loops (LL) and from some synthetic kernels. In the upper table, the mapping-invariant memory load model is applied, while the lower table uses the mapping-sensitive memory load model.

Kernel	Description	#Nodes n	#Edges m	ILP model for $p=6$ var's	constr.	time	ILP model for $p=8$ 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, second 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

Kernel	Description	#Nodes n	#Edges m	ILP model for $p=6$ var's	constr.	time	ILP model for $p=8$ var's	constr.	time
LL9	Integrate predictors	28	27	498	695	12:10s	667	917	—
LL10	Difference predictors	29	28	519	720	—	691	950	—
LL14	1D particle in cell, second loop	19	21	357	542	0:08s	475	716	1:15s
LL22	Planckian distribution	10	8	171	221	<0:01s	227	291	<0:01s
FIR8	8-tap FIR filter	16	22	327	563	—	435	745	2:43s
T-8	Binary tree, 16 leaves	31	30	555	770	5:22s	739	1016	0:19s
C-6	Cook pyramid, 6 leaves	21	30	435	760	—	579	1006	—

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 ($\epsilon_M \gg \epsilon_C$) are also given in Table 2.

Figure 3 shows a Pareto-optimal mapping for Livermore Loop 9 onto $p = 6$ SPEs where the maximum memory load is minimized. This was computed within 2 minutes. In contrast, for the same task graph with $p = 8$ (443 variables, 485 constraints) CPLEX could not find an optimal solution within 24 hours of optimization time. Comparing this to the case of merge trees above, we see that in comparison to general DAG

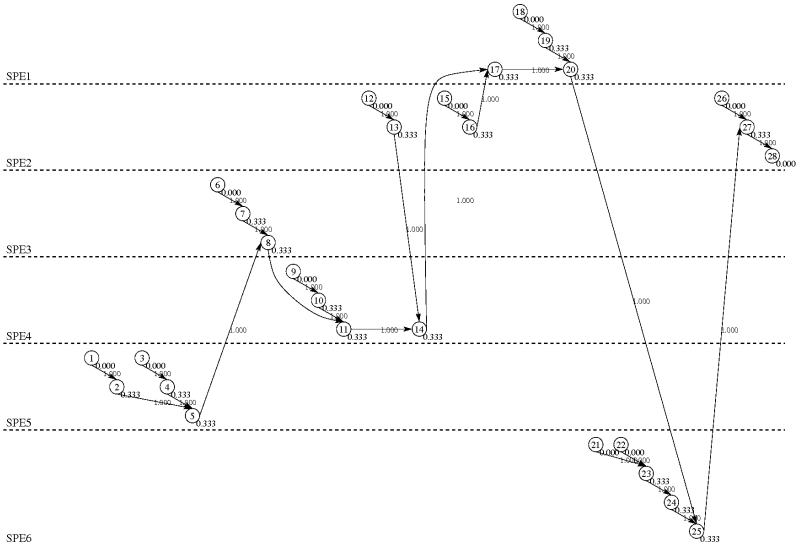

Fig. 3. A Pareto-optimal solution for mapping the dataparallel task graph of Livermore Loop 9 onto 6 processors with minimal memory load ($\epsilon_M \gg \epsilon_C$) using the mapping-sensitive memory load model, as computed by the ILP solver. Circles represent tasks and are annotated with normalized computational load, edges with normalized data rates.

topologies, much larger merge trees can be mapped optimally. Table 2 shows also that the optimization problems become more complex when applying the mapping-sensitive memory load model. Interestingly, the task graphs for the synthetic kernels FIR8, T-8 and C-6 show unexpected behavior in optimization time, where the smaller problem instances take significantly longer time than the larger ones. We have no explanation for this.

A discrete event simulation of the LL9 mapping on 6 SPEs (Figure 3) achieved a pipeline efficiency of close to 100%.

For small n, the task granularity as defined in the mapping problem gets too coarse such that the computational capacity of the Cell may not be fully exploited. Fortunately, in contrast to merge trees, dataparallel operations can easily be split by splitting the operand vectors. Hence, a workaround to this problem could be to split the task graph into 2 copies each working on half vectors only, thus halving the computational load of each node, and repeat the mapping step. A drawback is that this increases the problem complexity.

3.6 Heuristic Mapping Algorithm for Large Task Graphs

For large n and/or p where the ILP solver exceeds the time limit, we can apply the following divide-and-conquer heuristic: First, we compute a partitioning of the graph into two subgraphs of approximately equal sizes, with balanced memory load and a small accumulated volume of inter-partition edge weights. This can, for instance, be

done by running our ILP model above with $p = 2$. We divide accordingly the set of Cell SPEs into two halves, too. Now we solve the mapping problem for each partition recursively. As both the graph size and machine size are approximately halved in each subproblem, these should be computed considerably faster.

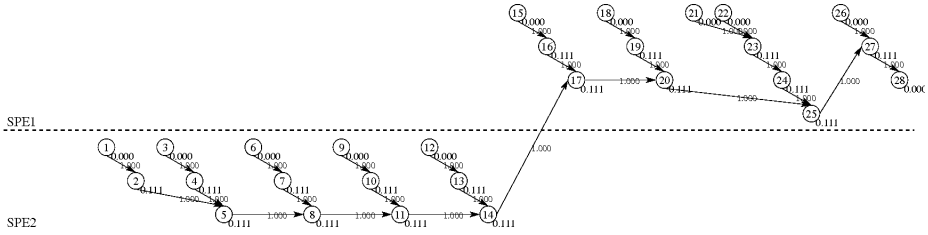

Fig. 4. ILP solution for partitioning the task graph of Livermore Loop 9 into two (thus $p = 2$) subgraphs, each to be mapped separately on a Cell SPE subset of size 4.

As an example, consider the case of Livermore Loop 9 with $p = 8$, for which the CPLEX solver could not find an optimal solution within 24 hours (Table 2) under either memory load model. For the partitioning, the ILP model for $p = 2$ was solved within 8 seconds, producing two subgraphs of 14 nodes each, connected by just a single edge, see Figure 4. The two ILP models for mapping the subgraphs optimally to 4 SPEs each were solved in less than a second. Both mapped subgraphs have a maximum memory load of 12, and the overall communication load is 1 + 5 + 5 = 11. There is, of course, no guarantee that the composed mapping for $p = 8$ is (Pareto-)optimal, as one with smaller maximum memory load or communication load may exist.

4 Related Work

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 (see e.g. Bokhari [13] for chain partitioning of linear pipelined task graphs into contiguous partitions) and trees (see e.g. Ray and Jiang [14]). The approaches for 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. This can 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 by a task merging pre-pass that coarsens the task granularity. See [15] for a recent survey and comparison. For Cell BE, Benini *et al.* [16] propose a constraint programming approach for combined mapping, scheduling and buffer allocation of non-pipelined task graphs to minimize the makespan.

Overlapping scheduling, which is closely related to *software pipelining*, 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 have been described e.g. by Hoang and Rabaey [17] and Ruggiero *et al.* [18]. Our method also belongs to this second category.

Hoang and Rabaey [17] 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.* [18] 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.

5 Conclusion

We have investigated 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 have formulated the general problem of mapping a task graph as an integer linear optimization problem. We have demonstrated our model with case studies from dataparallel code generation, merge trees in sorting, and FFT butterfly computations, and validated the mappings with discrete event simulations. 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 [19].

The ILP solver works well for small task graph and machine sizes, especially with the mapping-invariant memory load model. 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, especially if the more detailed mapping-sensitive memory load model should be used. We have proposed a divide-and-conquer based heuristic where the partitioning step is based on the ILP model for $p = 2$; other partitioning methods are possible and should be investigated, too. Moreover, developing approximation algorithms for the general case is an issue of future research. For the special case of merge trees, we have presented fast approximation algorithms in previous work [2,3].

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. 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)
3. Kessler, C.W., Keller, J.: Optimized on-chip pipelining of memory-intensive computations on the Cell BE. In: Proc. 1st Swedish Workshop on Multicore Computing (MCC-2008), Ronneby, Sweden. (2008)
4. Keßler, C.W., Paul, W.J., Rauber, T.: Scheduling Vector Straight Line Code on Vector Processors. In Giegerich, R., Graham, S.L., eds.: Code Generation - Concepts, Tools, Techniques, Springer Workshops in Computing (1992) 77–91
5. Akl, S.G.: Parallel Sorting Algorithms. Academic Press (1985)
6. Amato, N.M., Iyer, R., Sundaresan, S., Wu, Y.: A comparison of parallel sorting algorithms on different architectures. Technical Report 98-029, Texas A&M University (January 1996)
7. 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
8. 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
9. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley (1992)
10. Bisseling, R.: Parallel Scientific Computation – A Structured Approach using BSP and MPI. Oxford University Press (2004)
11. ILOG Inc.: Cplex version 10.2. www.ilog.com (2007)
12. McMahon, F.: The Livermore Fortran Kernels: A test of the numeric performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratories (December 1986) Source code: www.netlib.org/benchmark/livermorec.
13. Bokhari, S.H.: Assignment problems in parallel and distributed computing. Kluwer Academic Publishers (1988)
14. 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
15. 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
16. Benini, L., Lombardi, M., Milano, M., Ruggiero, M.: A constraint programming approach for allocation and scheduling on the CELL Broadband Engine. In: Proc. 14th Constraint Programming (CP-2008), Sydney, Springer LNCS 5202 (September 2008) 21–35
17. 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
18. 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)
19. 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)