SPECIAL ISSUE PAPER

Code generation for energy-efficient execution of dynamic streaming task graphs on parallel and heterogeneous platforms

Sebastian Litzinger | Jörg Keller

Faculty of Mathematics and Computer Science, FernUniversität in Hagen, Hagen, Germany

Correspondence
Sebastian Litzinger, Parallelism & VLSI Research Group, Faculty of Mathematics and Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany.
Email: sebastian.litzinger@fernuni-hagen.de

Summary
Streaming task graphs are high-level specifications for parallel applications operating on streams of data. For a static task graph structure, static schedulers can be used to map the tasks onto a parallel platform to minimize energy consumption for given throughput. We introduce dynamic elements into the task graph structure, thus specifying applications which adapt behavior at runtime, for example, switching from check-only to active mode. This in turn necessitates a runtime system that can remap tasks and potentially adapt their degree of parallelism in case of a dynamic change of the task structure. We provide a toolchain and evaluate our prototype with streaming task graphs both synthetic and from a real application. We find that we meet throughput requirements with <3.5% energy overhead on average compared with an optimal static scheduler based on integer linear programming. Runtime overhead for remapping is negligible and application runtime and energy are accurately predicted. We also outline how to extend our system to a heterogeneous platform.

KEYWORDS
dynamic task structure, energy-efficient code generation, parallel platform, streaming task graph

1 INTRODUCTION

Streaming task graphs are high-level constructs that are used to specify parallel applications, see, for example, StreamIT. They specify how the tasks interact, that is, which dependencies and which data flows occur between tasks. Streaming task graphs are helpful as more and more data are processed in the form of streams, for example, a stream of images from a camera processed one by one with a given throughput. The task graph might comprise four tasks, one initial task for contrast enhancement, two follow-up tasks for horizontal and vertical edge detection, and a final task that puts it all together such as checking if a certain object shape can be recognized from the edges. As the images arrive again and again, all tasks will be active simultaneously, however processing different images, see Figure 1. Thus, if there are implementations (either library or custom) of the tasks conforming to a given communication scheme, efficient application code for a parallel platform can be generated from the task graph description. The runtime system in its simplest form comprises a round-robin call to all tasks mapped to each processing core. To meet the throughput requirement, that is, reach a deadline for each round of task invocations, a scheduler maps the tasks to cores and determines suitable operating frequencies. As streaming applications often run on embedded systems, the energy consumption for each round should be minimized. As the task graph is statically given, a static scheduler can be used, and considerable effort can be put into finding a (near-)optimal mapping and task operating frequencies to meet the deadline with minimum energy. The tasks themselves can be parallel, so that a task might be mapped to more than one core. For example, if the workload of one task is much larger than for the others, a sequential implementation of this task may not be able to meet the deadline. As the degree of parallelism for each task is determined at compile time, the tasks are considered moldable.2
However, some applications demand some flexibility in the task structure. For example, depending on the type of image arriving, the applied filter type might vary, or an additional filter might be necessary. Thus, from time to time, a task might be modified. We propose how to extend static task graphs to cover such changes. As a consequence, the runtime system needs to cope with possible remapping of existing tasks, and adaptation of operating frequencies. Thus, tasks now can have different degrees of parallelism at different times, but are still moldable for each invocation. As the runtime overhead must be low, computing optimal mappings as in the static case is not feasible. To this end, we propose a heuristic mapper that is a hybrid of two (near-)optimal schedulers. The mapper reuses intermediate results from previous mappings when the task set to be scheduled is only varied slightly, such as increasing the workload of one task per round. We evaluate our prototype implementation by comparison to crown-optimal schedules for a benchmark set of streaming task graphs. On average, our schedules consume 2.8% more energy than crown-optimal schedules and are within 3.4% of the optimum, which seems a suitable compromise for the flexibility achieved. The time to remap a task set after modifying one task is small enough to invoke a remapping every other round without too much runtime overhead. Furthermore, we evaluate the accuracy of the scheduler's energy consumption predictions by executing the schedules in a prototype of the runtime system. The estimates are within 5% deviation from the experimentally determined values. Additional experiments with a real-world application see slightly more pronounced deviation from the scheduler's predictions but are still within 10% of the actual measurements at all times. Finally, we sketch how to extend our system to a heterogeneous platform, as such platforms are common among embedded systems.

The remainder of this article is structured as follows. In Section 2, we give background information on energy-efficient stream processing and related work. Section 3 introduces our concept of task graphs with dynamic elements. In Section 4, we describe our framework implementation. In Section 5, we present the dynamic mapper and remapper for tasks. Section 6 reports on experiments with both synthetic benchmark task sets and an example real-world streaming application. In Section 7, we sketch how to extend our approach to heterogeneous systems such as ARM's big.LITTLE, and in Section 8, we give conclusions and an outlook on future work.

2 | BACKGROUND

2.1 | Streaming task graphs

A task graph is a directed acyclic graph where each node represents a task from an application, and is annotated with the task workload. Each edge represents a dependency, that is, a task can only start if all of its predecessor tasks have completed. Typically, an edge also means a data transfer between a task and its successor task, which happens upon completion of the task. Figure 1 (left) depicts an example task graph with four nodes. In a classic model, where a uniform processing speed of the underlying platform is assumed, the workload is equivalent to runtime. However, for processing cores with frequency scaling, the task workload is given as the number of cycles, so that the runtime at frequency $f$ can be given as $t = w/f$. We assume that each task is executed at only one frequency. A streaming task graph is a model where a stream of data entering the task graph at the source node is processed by the task graph. As the stream of data is normally in the form of a sequence of data chunks, the task graph (processing one data chunk) is repeatedly invoked in the form of a streaming pipeline. If data chunks arrive with a certain rate, then this poses a throughput requirement on the task graph execution. Figure 1 (right) depicts this situation. The repeated invocation breaks execution into rounds, where each round comprises all tasks, yet from different task graph invocations, that is, independent of each other. The minimum throughput requirement implies a maximum length of the round, that is, a deadline $M$.

2.2 | Implementing streaming task graphs

Each task of a streaming task graph might be parallelizable in itself. Thus, there are two levels of parallelism when executing a streaming task graph: parallelism within tasks and concurrent execution of tasks. Let $t(q)$ be the number of cycles needed per core when the task is processed on $q$ cores, that is, $t(1) = w$. The parallel efficiency of the task is defined as $e(q) = t(1)/(q \cdot t(q))$, that is, $e(1) = 1$. The number of cores $q$ on which a task is executed is...
called the degree of parallelism or width. The maximum width $W$ of a task might be less than $p$, the machine size, or even 1 if the task is sequential. If the number of cores used might change during execution of the task, the task is called moldable, otherwise it is called malleable.\(^2\)

A processing core executing a task consumes power, mainly depending on operating frequency and instruction mix (if we assume that temperature is kept constant and operating voltage is always set at the minimum value possible for the operating frequency). The energy consumed is power integrated over time, which simplifies to the product of power consumption and runtime for a task if we assume a constant instruction mix and frequency. Tasks might be classified into task types\(^1\) with similar power profiles. Then task energy can be computed as $E = (t_{\text{(q/f)}} \cdot P(f, T) \cdot q$, if $P(f, T)$ denotes the power consumption at frequency $f$ and task type $T$. The energy for one round of streaming task graph execution is the sum of all task energies. While cores also draw power when they are idle, this can either be ignored if idle power $P_{\text{idle}} \ll P(f, T)$, or one can subtract idle power from $P(f, T)$. In this case, total energy is $E + P_{\text{idle}} \cdot p \cdot M$, but any scheduling decision only influences $E$.

Given a parallel platform of $p$ processing cores with their frequency and power profiles, executing a streaming task graph application demands to allocate suitable processing resources, that is, width, and operating frequency to each task, and map the $n$ possibly parallelizable tasks onto the cores and in time such that all tasks are executed prior to the deadline and the energy consumption is minimized. If the streaming task graph is static, a static schedule can be computed prior to execution. As this computation is done once and offline, a considerable effort can be put into finding an optimal or near-optimal schedule. However, if the streaming task graph can vary during execution, dynamic scheduling is necessary.

The crown scheduler\(^4\) uses an integer linear program to map a set of moldable tasks onto a parallel machine with multiple frequency levels. To reduce the number of possibilities, only allocations $q$ that are powers of 2 are permitted. Also, the cores are arranged in groups: one group of width $p$, followed by two concurrent groups of size $p/2$, and so on. The restriction to map tasks only to groups further reduces the number of possibilities to map tasks onto cores and arrange them in time, as the groups are executed in decreasing order of width.

The approach by Sanders and Speck\(^3\) maps a set of malleable tasks onto a parallel machine with frequency scaling, giving them continuous widths (also called allocations) $p_j + x_j$, where $p_j$ is integral and $x_j$ between 0 and 1. The meaning is that the task runs on $p_j + 1$ cores for time $x_j/M$ and on $p_j$ cores for the rest of the time. Thus, a task is only parallelized if $p_j \geq 1$. Each task gets $p_j$ cores that it uses alone, the $x_j$ "pieces" of the tasks are mapped to the remaining cores, which might necessitate to preempt a task and continue it later on a different core. Figure 1 (right) shows a schedule for four tasks on six cores with allocations 2.0, 1.8, 1.3, and 0.9, respectively. The smallest task is preempted and migrated. A task’s execution frequency is chosen from a continuous range for minimum energy of the chosen allocation. Energy only depends on the allocation, which is chosen such that the steepness of the energy curves for all tasks is identical while the sum of the allocations equals $p$, the number of cores. This point is found by starting with interval $(-\infty, 0)$, repeatedly choosing the middle of the interval as derivation value and computing the corresponding allocations by inverting the energy derivation function, and then halving the interval. This procedure seems to be an example of optimization with constraints solved by Lagrange multiplier,\(^9\) where the repeated interval halving is a standard solution technique. No task types are considered and the power and efficiency curves must meet certain criteria. In Reference 9, use of this scheme for static scheduling of moldable tasks and discrete frequency levels is investigated.

\section{Related work}

In Reference 5, the DRAKE framework is presented which uses tasks specified by the user. The task graph is specified in GraphML, the task implementations are given as C/C++ code. However, the tasks cannot be modified at runtime, and different task types, that is, differences in instruction mix leading to different power consumption (see below) are not considered. The DRAKE framework uses the crown scheduler\(^4\) to compute a static mapping of tasks to cores. The StreamIT framework extracts tasks and their parallel efficiency functions from user programs.\(^1\) However, also here tasks are static.

An energy-aware dynamic scheduler for task-based parallel programs is presented in Reference 10 with a focus on tasks with dependencies. While tasks without dependencies are possible, the schedule is for a sole execution and thus incremental changes to the workload are not covered. Also, a combination of application runtime and energy consumption is optimized, while we target minimization of energy for a given application deadline. Furthermore, scheduling time (see Figure 14 in Reference 10) for 1000 tasks on 10 processors is on the order of 50 ms, that is, not really suitable for regular changes. In Reference 11, a dynamic scheduler is presented that minimizes response time and simultaneously improves energy consumption and degree of fault tolerance. By contrast, we target task sets with a common deadline.

\section{STREAMING TASK GRAPHS WITH DYNAMIC ELEMENTS}

We derive our task graph specification from Reference 5, which in turn is based on GraphML. Thus, our specification can be considered as a domain specific language (DSL) of the embedded type.\(^12\) As usual for rather specialized domains, we have to find a compromise between user comfort and tool complexity, especially as the tool support so far is an academic prototype.
The task graph specification contains the deadline to be met by the task graph, the set of nodes, and the edges between nodes. For each node, the workload, the maximum degree of parallelism (max. width), the parallel efficiency for each width, the task type and the function call implementing the task is given (cf. Figure 2). In addition to the task graph specification itself, source code for each task function must be given. The first call parameter is the width at which the task is to be executed in a specific invocation. The return value must be an integer and shall be zero to denote normal completion, positive if the task announces to toggle its mode (see below), and negative in case of an error encountered during task execution. The task workload, the efficiency function and the task type can either be derived analytically or via manual profiling by the user, as an automatic profiling by the compiler is currently out of reach. Thus, we target applications with computational tasks that have predictable runtimes. Tasks with stochastic behavior might be treated in a future extension, also memory-bound tasks can be handled by extending the efficiency function.

We extend the task graph specification by allowing each task to be in either of two modes (cf. Figure 2). The modes can differ in their workloads, that is, also task type, their maximum widths, and their parallel efficiencies. This form of extension has the advantage that still all tasks are known to the system from the beginning, that is, no task is generated completely anew during runtime, and thus the formulation can remain as static as before, while addition and removal of tasks can be handled by this scheme, if one mode assumes the task is inactive (removed) and only forwards its input to its output. Addition of a task then is the switch to active mode. Also, each task will participate in each invocation of the task graph execution. The actual mode is given to the task function as the second call parameter.

Our initial idea behind this extension is that a task may start in a check-only mode where it basically copies input to output and checks if the input data is of sufficient quality. If the buffer management system is suitably arranged (cf. Section 4), actual copy can be avoided, so that this operation is cheap. If the input data quality is not good, the task signals by its return value that it wants to toggle into active mode. In that mode, the task performs its intended processing function on the input data (such as filtering or contrast enhancement in image processing applications), outputs processed data, and checks if it could go back to check-only mode.

Naturally, more types of dynamic change of mode are possible. Thus, a task x that only implements one mode itself could signal by a positive return value y that another task y ≠ x should toggle its mode. This could for example be used if a follow-up task is assuming unsorted data and does initial sorting. If future data will be presorted, a predecessor task could inform its follow-up task that it can skip the sorting and a merge of presorted streams is sufficient, which would make the follow-up task much cheaper. While such information could also be transported to the follow-up task directly with the communicated data, signaling it this way has the advantage that the runtime system will learn immediately of the mode change and can plan processing resource allocation accordingly. Also, a task might implement more than two modes, so that it would toggle through all modes before returning to its initial mode.

To have a systematic description which task influences which other tasks’ modes, a control-flow graph can be created as in Akioka (cf. his Figure 3), although his focus was not dynamic behavior of tasks.

Falk et al. allow only a part of the task graph to be dynamic, and the rest static, to be able to do most scheduling decisions prior to execution. In contrast, we allow each task to have multiple modes, but assume that only a fraction of the tasks switches modes in a single round.

Implementing a streaming task graph framework can be done on several levels of complexity. In the first run, the structure of our implementation draws from DRAKE (while the implementation itself is independent and completely new, with the notable difference that tasks may exhibit different behavior at different times, and thus the static scheduling done in DRAKE has been replaced by a dynamic mapping and remapping of tasks. The general workflow is depicted in Figure 3. The parts are described below. Such an implementation has the advantage that frequently needed parts such as buffer management can be more mature than if written from scratch for each application, thus saving on development and debug time and improving performance.
When compiling the task graph specification, a runtime system is created. This runtime system maps each task to one or more cores (see Section 5), depending on whether the task is parallelizable and whether enough resources are available. We initially planned to use OpenMP with an outer parallelized loop over the parallelized (plus set of sequential) tasks, and second-level parallelization for each group. However, nested parallel regions must be of equal sizes, while the widths of the parallelized tasks can differ. Thus, we create $p$ POSIX threads which we pin to the cores, and have a task of width $w$ in the task lists of $w$ threads.

Each task gets a description structure comprising the system-wide task ID, a pointer to the function implementing the task, its mode (such as active or check-only), the frequency level it should be run on, and the number of cores it should use. Then, the runtime system has each core execute its assigned tasks in a round-robin manner, see the code in Figure 4. The overhead for task execution is low, especially if tasks are sorted according to operating frequency, to minimize the number of frequency changes. If a task function returns a nonzero value, then this indicates a change of mode or an error, and is checked. When such an event leads to the necessity to adapt the mapping and/or operating frequencies of tasks, a remapping is invoked (see Section 5).

If the remapping is fast enough (at least if amortized over several rounds until the next change), it can simply be performed between two rounds, and the round deadline is shortened accordingly. Otherwise, the new mapping must be computed concurrently to task execution. While the remapping is going on, the tasks are still executed with the current mapping, however at increased frequencies to have some processing power for the remapping. For reasons of readability, this is not shown in Figure 4.

Tasks communicate with each other, but users should only need to specify reads from task input buffers and writes to task output buffers. Thus, when compiling the specification, a buffer management system (e.g., library code plus initialization derived from XML) is created that allocates and initializes a buffer for each task graph edge. A task can query the buffer management structure for its input and output buffers, and gets pointers to the buffer structures. Then a task can invoke read functions for input buffers and write functions for output buffers. Read functions are available in blocking and nonblocking form. The latter variant is necessary as sometimes, for example, during the start phase of the application, an input buffer might be empty. In this case, the task should be skipped in this round. The buffer management library also provides a function to copy from input to output buffer in a fast way, without actually moving data, but only a pointer to the location of the data. This is helpful in check-only mode. This quick copy is possible by distinguishing between the data that are forwarded (allocated memory) and the buffer data structure, which only contains a pointer to the data itself, and some metadata. This allows tasks with only one input and one output buffer to modify input data in active mode and forward it to a successor task without new memory allocation. On the downside, a task with one input and two output buffers will have to duplicate its input data even in check-only mode before forwarding it via the two output buffers.

Please note that getting input into the task graph, such as getting images from a cam, is not the duty of the buffer management, but done in the source task. Similarly, writing results to some output file or database is the duty of the sink task.

5 MAPPING AND REMAPPING TASKS

Upon the start of the application, a mapping of the tasks to cores (including determining their widths and operating frequencies) is performed. Every time a task changes its behavior, a remapping is initiated. This remapping must be fast enough that it can be completed before the next round...
starts, or within one (or a few) rounds, during which tasks run with increased frequencies to free some processing resources for the remapper. In the following, we describe the complete mapping, and point out which parts of the mapping can be skipped for a remapping. Thus, as we can use some intermediate results from the initial mapping, a remapping normally can be done much faster than the initial mapping, thus reducing the overhead at application runtime.

The mapping consists of assigning each task \( j \) a number of processors \( q_j \); it is run on as well as a corresponding operating frequency \( f_j \), such that all tasks are completed before the common deadline \( M \) and the energy consumption is minimized. The mapper borrows from a very fast approach\(^3\) to parallelize tasks only if they allocate more than one core for the complete time till the deadline, and to find a common steepness for all tasks’ \( j \) energy functions (depending on \( q_j \)) such that the processor allocation \( \sum q_j \) equals \( p \) and the deadline can be met. As this approach only parallelizes a task if it allocates one core for the complete time till the deadline, the mapping of the tasks reduces to an allocation for parallel tasks (their mapping is trivial as task 0 gets the first \( q_0 \) cores, task 1 gets the next \( q_1 \) cores, etc.). Only the sequential tasks must be mapped, or more correctly distributed, onto the remaining cores. But even this can be simplified, see below.

We assume that the mapper knows power values for all task types and frequencies on the targeted hardware platform. Those values can be derived from microbenchmarks as in Reference 7 or could be derived from more abstract models. As tasks are moldable, the energy function is discretized, and only core counts \( q_j \) that are powers of 2 are permitted, which is an idea taken from Reference 4. For tasks that remain sequential, the time that might be used is restricted to \( 2^{-i}M \), denoted by \( q_j = 2^{-i} \). If several cores are used to execute sequential tasks, load balancing problems are avoided by using powers of 2, as sequential tasks can be sorted in descending order of width, and grouped such that their widths sum to 1. As the resulting mapping is a special crown schedule if the parallel tasks are assigned in decreasing order of width (a task with \( p_j \geq 1 \) is the only task in its group, and smaller groups on the same cores are empty), we can use crown-optimal schedulers, extended with the additional constraints from our mapper. While using an ILP-based scheduler is not feasible in real use, it can be used to find out how close the heuristic scheduler is to the optimum.

The smallest exponent yielding a possible width is calculated as \( \left\lfloor \log_2 \frac{W_j}{M} \right\rfloor \), which guarantees that an allocation with \( \sum q_j \leq p \) exists—given that the deadline is not overly tight. At the same time, the exponent clearly cannot be larger than \( \log_2 p \). For example, when scheduling a set of 10 tasks on a machine with four cores, the smallest exponent would be \( -\left\lfloor \log_2 \frac{10}{4} \right\rfloor = -2 \) and the largest exponent \( \log_2(4) = 2 \), resulting in possible allocations of \( 2^{-2}, 2^{-1}, 2^0, 2^1 \), or \( 2^2 \) cores.

Since we operate with a small set of discrete possible allocations—as opposed to the infinitely many possible allocations for malleable tasks\(^3\)—the originally proposed interval halving technique to obtain the optimal steepness value does not work well in our case, so we have to come up with a different approach. Generally speaking, if we assume an energy function which decreases monotonically with increasing processor allocation, the more energy can be saved the smaller the steepness value is (or, as under the assumption just stated all steepness values are \( \leq 0 \), the larger the steepness’s absolute value is) when moving from a particular allocation to the next possible one. The basic concept of the scheduler must therefore lie in extending the processor allocation for tasks where it pays off most energy-wise, after an initial allocation has been computed and while \( \sum q_j < p \), that is, there are resources left. To achieve this, our scheduler proceeds as follows: Initially, the frequency level leading to the lowest energy consumption is determined for each task and allocation.

Figure 5 shows the energy curve and the corresponding steepness curve for an example compute-bound task with workload \( w = 10^9 \) cycles and optimal parallel efficiency \( e(q) = 1 \), deadline 1 s, and for a synthetic machine with power profile \( P(f) = f^3 \) (W/GHz\(^3\)), depending on continuous allocation. As our allocations are discrete, only the marked points can be taken as width values. If there are two tasks, we start with the points to the left, and increase the allocation for the task with the higher steepness value to the next possible allocation, as long as the sum of the allocations does not exceed \( p \).

Note that an allocation might be infeasible if the task cannot be completed prior to the deadline even at the highest frequency level, or if \( q_j > W_j \). These allocations are excluded, as are those which see an increase in energy consumption compared with smaller task widths, that is,

**Figure 5** Example energy consumption and energy derivation curves depending on allocation
those not satisfying $E_j(f^*, q) < E_j(f^*, q')$ for $q' < q$, where $f^*$ is the energy-optimal frequency for the respective allocation and task. Such a scenario is perfectly conceivable as parallelizing a task induces a certain overhead which may not be compensated by the potentially lower energy consumption due to lower possible operating frequencies. For the remaining valid allocations, we approximate corresponding steepness values by interpolation as

$$st_j(q) = \frac{E_j(q) - E_j(q^{\text{prec}})}{q - q^{\text{prec}}},$$

where $q^{\text{prec}}$ is the allocation immediately preceding $q$. For the first valid allocation, we set steepness to $-\infty$. In principle, building such a steepness table is required for each individual application. One may however work on some level of abstraction and create profiles for various task types primarily featuring a certain operation (or mix of operations) like floating point multiplications or memory accesses. In that case, the compiler would have to determine a task’s type via code analysis, whereupon an adequate steepness table could be generated based on this information and the task’s other parameters. Then, the task/allocation pairs are gathered in a list sorted by ascending steepness. The first $n$ pairs represent the smallest values $q_j$ facilitating a feasible allocation for the respective task. Now, one traverses the remaining list and for each entry checks whether under the new potential allocation, $\sum q_j \leq p$. If this is the case, the new allocation is assumed by changing the allocation of the task mentioned in the current entry to the new value. If accepting the current entry’s allocation would lead to $\sum q_j > p$, it is ignored and one proceeds to the next entry. When the end of the list is reached or one encounters an entry with a steepness value of 0 (signifying stagnation of energy consumption when altering the allocation), the final allocation has been ascertained. If steepness increases monotonically with $q$, one can already stop traversing the list as soon as $\sum q_j = p$, as each modification of an allocation is in fact an extension. If steepness does not increase monotonically with $q$, one has to check whether energy consumption actually decreases when assuming a new allocation, since one might be attempting to move toward smaller $q$ values. Algorithm 1 summarizes the procedure detailed thus far.

Algorithm 1. Heuristic determination of core allocations for each task

\begin{verbatim}
input : $t_j(1), W_j, e_j, P(f, T_j)$
output: feasible allocation values for all $n$ tasks
compute smallest possible allocation exponent $-\left\lceil \log_2 \frac{n}{p} \right\rceil$;
compute largest possible allocation exponent $\log_2 p$;
foreach task $j$ and possible allocation $q$ do
    determine frequency level $f^*_j$ leading to lowest $E_j$;
end
foreach task $j$ do
    exclude allocations $q$ for which $j$ cannot be completed until the deadline, or where $q > W_j$;
end
foreach task $j$ do
    exclude allocations $q$ not satisfying $E_j(f^*_j, q) < E_j(f^*_j, q')$ for $q' < q$;
end
foreach task $j$ and corresponding first valid allocation $q^{(0)}_j$ do
    set $st_j(q^{(0)}) = -\infty$;
end
foreach task $j$ and remaining valid allocations $q$ do
    compute $st_j(q) = \frac{E_j(q) - E_j(q^{\text{prec}})}{q - q^{\text{prec}}}$;
end
gather all task/allocation pairs in a list sorted by ascending $st_j(q)$;
determine the initial allocations from the first $n$ list entries;
foreach remaining list entry do
    if $E$ decreases under new allocation and $\sum q_j \leq p$ then
        adopt new allocation;
end
end
\end{verbatim}
Let us consider a tiny example for a task set consisting of two tasks with $w_0 = 7$, $w_1 = 18$, $W_0 = W_1 = 4$ on a machine with $p = 4$ cores and possible allocations $2^{-2}, \ldots, 2^2$. We assume the energy table looks like Table 1. It shows that task 0 requires a minimum allocation of 0.5 cores while task 1 needs at least a whole core of its own to complete before the deadline.

Table 2 is the corresponding steepness table. Note that allocations of two and four cores have been eliminated for task 0, and task 1 shall not run on four cores since the energy consumption rises again for these allocations, cf. Table 1. The smallest $q_j$ yielding a feasible allocation are $q_0 = 0.5$ and $q_1 = 1$. These values are designated as initial allocations, cf. Figure 6, which also contains all other valid allocations. As the current allocation is $q_0 + q_1 = 1 + 0.5 = 1.5 \leq p$, the scheduler can proceed to the next value in the sorted steepness list, cf. Figure 7. We now have $q_0 = 0.5$ and $q_1 = 2$ for a total allocation of $2.5 \leq p$. Therefore, the new allocation can be confirmed (see Figure 8). Further traversal of the steepness list leads to extending $q_0$ from 0.5 to 1, cf. Figure 9. As we now have $\sum q_j = 3 \leq p$, this change of allocation is approved as well. Figure 10 shows the current situation, which is also the final one since the end of the sorted steepness list has been reached and the allocation algorithm terminates.
For a remapping, we already have the lists of possible allocations (and their energy consumption) for each task, if we have computed these allocations for each task and mode initially. Thus, we need only traverse the lists again to find allocations with similar steepness values of the energy function, and $\sum q_j \leq p$. We can even skip part of this traversal. If, for example, only one task switches to a mode with lower workload, then other tasks might only get larger allocation than before, that is, we can start with the current allocations in the list. If only one task switches to a mode with higher workload, we start from current allocations in reverse direction, as some tasks might get lower allocation than before.
6 | EVALUATION

In our experiments, we first focus on the evaluation of the scheduler elaborated on in Section 5. To this end, we have conducted a set of experiments with synthetic task sets of varying cardinality. The resulting energy consumption when scheduled via the proposed technique is compared with the energy consumption of a crown-optimal schedule. Moreover, the execution time of various steps in the scheduler is recorded, since adjustments to the schedule during a streaming application’s execution may be required (as explained in Section 3). In these situations, the scheduler must quickly provide an updated schedule.

In a second set of experiments, we intend to learn about the practical applicability of our approach. Here, we use an example streaming application performing image processing and concentrate on the scheduler’s ability to accurately predict task runtimes and energy consumption. Furthermore, the runtime overhead of dynamic remapping plays an important role in these experiments as well. To obtain the required information, we have deployed an implementation of the application and the runtime system sketched in Section 4 on an Intel Xeon CPU.

6.1 | Experiments with synthetic task sets

For our first set of experiments, we have created 40 synthetic task sets containing 10, 20, 40, or 80 tasks (10 for each cardinality). Workloads for each task are integers chosen randomly from [1, 40] and maximum widths are integers from [1, 4], both based on a uniform distribution. For our current purposes, the runtime system assumes homogeneous tasks performing floating point operations. As a paradigmatic workload, we have adopted the run_dmul function shown in Figure 11 from the epEBench benchmark, and stipulated that one unit of workload be equal to 10,000 loop iterations of the given function. On the hardware side, we have chosen a machine featuring an Intel Xeon E5-1620 v3 processor with four physical cores and \( F = \{1.2, 1.4, 1.5, 1.7, 1.9, 2, 2.2, 2.3, 2.5, 2.7, 2.8, 3, 3.2, 3.3, 3.5\} \) GHz to host the runtime system. To enable precise scheduling on the targeted platform, we had to gather information on core power consumption as well as runtime when executing the specified function at various frequency levels.

We have determined power consumption values by measuring core energy consumption via the sysfs interface and dividing by runtime. This has been done for each combination of core frequency levels (excluding turbo mode, and without regard to core order; that is, the combinations of core frequency levels (0,2,4,9) and (2,0,9,4) are considered equal, for instance), which can, for example, be facilitated by choosing the userspace scaling governor and manually setting the individual core operating frequencies. Since the E5-1620 v3 offers 15 regular frequency levels, and a core can potentially be idle (in that case we have assumed it to be running at the lowest frequency level under no load), we have conducted \( \left( \frac{15^2}{2} \right) = 3876 \) individual experiments. Each of these consisted of thrice measuring average power consumption when executing run_dmul for 5 s and averaging over these values, with 10 s of idle time between two consecutive measurements, and another 10 s in between experiments. In the end, we are

```c
void* run_dmul(double workload) {
    int i;
    volatile double x=1.123, y=0.23, z=6.32;
    int numruns = (int)(10000*workload);
    for (i=0; i<numruns; i++) {
        x*=1.121;
        y*=1.122;
        z*=1.123;
        x*=1.124;
        y*=1.125;
        z*=1.126;
        x*=1.127;
        y*=1.128;
        z*=1.129;
        x*=1.130;
        y*=1.131;
        z*=1.132;
        x*=1.133;
        y*=1.134;
        y*=1.135;
        x*=1.136;
        y*=1.137;
        z*=1.138;
        x*=1.139;
        y*=1.140;
        z*=1.141;
        x*=1.142;
        y*=1.143;
        z*=1.144;
        x*=1.145;
        y*=1.146;
        z*=1.147;
        x*=1.148;
        y*=1.149;
        z*=1.150;
    }
    return NULL;
}
```

FIGURE 11 | Code of the run_dmul function adopted from Reference 16
interested in a power consumption figure for a core running at a specific frequency, which contributes to the processor’s overall power consumption modeled as

\[ P_{\text{total}} = \sum_{i=0}^{n} P_i + P_{\text{idle}}, \]  

(1)

that is, the sum of the individual cores’ power consumption values and the idle power \( P_{\text{idle}} \). From our experiments, we receive a massively overdetermined system of linear equations. To obtain the values we seek, we have employed the method of least squares to our experimental data.

In order to provide the scheduler with reliable runtime estimates, we have executed \texttt{run_dmul} with 10,000 loop iterations on all four cores. For each frequency level, this was done 100 times. Since there is a noticeable value spread, we have chosen the median at each frequency level as runtime value for the scheduler.

The tasks’ parallel efficiency is defined as in Reference 17:

\[
e_{j}(q) = \begin{cases} 
  1 & \text{for } q = 1, \\
  1 - 0.3 \frac{q}{W_{j}}^2 & \text{for } 1 < q \leq W_{j}, \\
  0.000001 & \text{for } q > W_{j}.
\end{cases} 
\]

This is not completely accurate for tasks based on \texttt{run_dmul}. For such tasks, the parallel efficiency will be dependent on the number of iterations, and might be a bit higher than \( e(2) = 0.88 \) and \( e(4) = 0.7 \) as given in the formula.

The deadline \( M \) is computed similar to Reference 4:

\[
M = 0.9 \cdot \frac{\sum_{j} w_{j} \cdot f_{\text{max}} + 2 \sum_{j} w_{j} \cdot f_{\text{min}}}{2}.
\]

Here, \( f_{\text{min}} (f_{\text{max}}) \) is the minimum (maximum) operating frequency. The factor of 0.9 serves to tighten the deadline, which makes scheduling more difficult especially for the heuristic approach. For even tighter deadlines, the heuristic does not produce a feasible solution in all cases, so this is the tightest deadline facilitating a comparison of the results.

To assess the quality of the schedules produced by our heuristic scheduler, we compare them to crown-optimal schedules computed offline by an integer linear program realizing integrated crown scheduling. As a further reference, we have attempted to compute optimal schedules—even dropping the crown restrictions—with an ILP from Reference 18. Especially for the larger task sets, we could not obtain a feasible solution prior to timeout, and even for most smaller task sets, an optimal solution could not be reached. Therefore, we provide theoretical energy consumption values based on the best bound values at the point of timeout, that is, values which cannot be undercut even by an optimal solution. We have deployed Python implementations of the crown scheduler and the optimal scheduler using the Gurobi 8.1.0 solver and the \texttt{gurobipy} Python module. The scheduler introduced in Section 5 has been implemented in C. All computations were carried out on an AMD Ryzen 7 2700X, where the crown scheduler and the optimal scheduler operated under a 5 min (wall clock) timeout.

Table 3 shows the energy consumption obtained via the proposed scheduling technique and via crown scheduling, as well as the optimal scheduler’s best bound values at timeout. The values are averaged over all task sets of a particular cardinality. As becomes obvious, one must accept only a minor energy overhead of 2.8% on average relative to crown scheduling, which remains nearly constant over all examined task set sizes. Worst

| #Tasks | 10   | 20   | 40   | 80   | Total |
|--------|------|------|------|------|-------|
| Avg. E optimal best bound (J) | 0.497 | 0.946 | 1.916 | 3.780 | 1.785 |
| Avg. E crown (J) | 0.498 | 0.952 | 1.928 | 3.803 | 1.795 |
| Avg. E hybrid (J) | 0.516 | 0.977 | 1.983 | 3.909 | 1.846 |
| Avg. E crown versus opt. b.b. | 1.003 | 1.006 | 1.006 | 1.006 | 1.006 |
| Worst E crown versus opt. b.b. | 1.006 | 1.006 | 1.006 | 1.006 | 1.006 |
| Avg. E hybrid versus crown | 1.034 | 1.026 | 1.029 | 1.028 | 1.028 |
| Worst E hybrid versus crown | 1.138 | 1.044 | 1.039 | 1.041 | 1.138 |
| Avg. E hybrid versus opt. b.b. | 1.038 | 1.032 | 1.035 | 1.034 | 1.034 |
| Worst E hybrid versus opt. b.b. | 1.138 | 1.050 | 1.045 | 1.048 | 1.138 |
case values are 13.8% for task sets of size 10; for larger task sets, there is no case where the relative energy overhead was >5%. With regard
to the optimal scheduler’s best bound values, the energy overhead amounts to 3.4% on average, that is, the energy consumption when executing
the heuristic’s schedules is within 3.4% of the optimum on average. Here, the worst case is 13.8% overhead as well for a task set of size 10, with consider-
ably lower worst case values for larger task sets. We also experience a confirmation of the claim in Reference 18: in the cases examined here, the
crown scheduler’s results are indeed near-optimal (≤0.6% worse than the optimum on average).

As the purpose envisaged demands a rapid execution of the scheduler itself, it will be interesting to look at the performance figures in detail.
For the crown scheduler, we give the execution times of the whole scheduling application (sum of user and system times as measured by the pro-
cess_time() function of Python’s time module). For the newly introduced scheduling technique, we have recorded the time to compute the
energy consumption values at the most favorable frequency level for each allocation and task (eval), the time to compute the steepness table (etab),
the time to sort the list of steepness values (stsort), and the time to move through the remaining steepness list entries after the initial (i.e., first
valid) allocation for each task has been determined (stlist). Finding the initial allocations simply means traversing the sorted steepness list up to
the first valid value, which can be achieved with negligible effort and therefore is ignored in the following considerations. These measurements were
obtained via the clock() function and subsequent division by CLOCKS_PER_SEC. Tables 4 and 5 show the respective values.

From Table 4 it can be gathered that the crown scheduler—although computing a close to energy-optimal schedule—cannot prevail in the current
devour as execution times are by far too high. They average at several hundred milliseconds for small and large task sets, whereas for medium-sized
task sets, computation of a schedule may take seconds of CPU time. Moreover, dispersion is quite pronounced in these cases.

Table 5 illustrates that the major computational effort regarding the proposed scheduling technique stems from the computation of the energy
values at the most favorable frequency level for each allocation and task, as well as sorting of the steepness value list. Here, it should be pointed out
that computation of both the energy values and the steepness values can be performed statically (since all tasks that may have to be executed in the
course of the streaming application are known beforehand). For schedule modifications at runtime, steepness values for the tasks to be added must
be merged into the (already sorted) steepness value list, which certainly requires less effort than sorting the whole list. Subsequently, the steepness
list must be traversed to determine the new processor allocations. These two steps (stsort and stlist) can be carried out in <1 μs per task on average,
cf. Table 5. As it becomes clear during one scheduling round which tasks are to be scheduled additionally or differently for the next round, this time
will be short enough to adjust the schedule accordingly with low overhead if the scheduling round, that is, execution time for the tasks, has a length
of >100 μs. For shorter rounds, the change in task behavior should be more seldom to avoid excessive runtime overhead.

In order to evaluate the accuracy of our hybrid scheduling technique in terms of energy consumption prediction, we have implemented a pro-
totype of the runtime system conceived in Section 4. The schedules computed by the hybrid scheduler have been executed on the runtime system
prototype for 1000 rounds each, and for each round, energy consumption has been measured. For our experiments with synthetic task sets, we
have not performed dynamic schedule adaptations with the runtime system. Table 6 shows the energy consumption values predicted by the sched-
uler compared with the ones measured by the runtime system. The provided values are averaged over the 10 task sets for each task set size. As one
can see, the scheduler’s energy consumption predictions surpass the real values by 0.9%–2.4% on average (with marginally lower errors for larger
task sets), for a total average of 1.5%. In some cases the scheduler predicted energy consumption lower than the actually measured values, down to
98.6% of the experimental result. The worst energy consumption prediction for a single task set differs 5.5% from the experimentally determined
value. Overall, the scheduler is capable of predicting the energy consumption caused by a schedule’s execution with high accuracy.

| Tasks | 10 | 20 | 40 | 80 | Total |
|-------|----|----|----|----|------|
| Avg. time(s) | 0.589 | 3.730 | 0.393 | 0.262 | 1.244 |
| SD | 0.235 | 6.362 | 0.414 | 0.092 | |

**Table 4** Execution times for the crown scheduler, averaged over all task sets of a size

| Tasks | 10 | 20 | 40 | 80 | Total |
|-------|----|----|----|----|------|
| Avg. t eval (μs) | 22.80 | 53.40 | 114.70 | 242.60 | 108.38 |
| Avg. t eval/task (μs) | 2.28 | 2.67 | 2.87 | 3.03 | 2.71 |
| Avg. t etab (μs) | 1.70 | 2.50 | 4.60 | 8.50 | 4.33 |
| Avg. t etab/task (μs) | 0.17 | 0.13 | 0.12 | 0.11 | 0.13 |
| Avg. t stsort (μs) | 7.50 | 16.60 | 38.30 | 73.00 | 33.85 |
| Avg. t stsort/task (μs) | 0.75 | 0.83 | 0.96 | 0.91 | 0.86 |
| Avg. t stlist (μs) | 1.10 | 2.40 | 5.00 | 8.30 | 4.20 |
| Avg. t stlist/task (μs) | 0.11 | 0.12 | 0.13 | 0.10 | 0.11 |

**Table 5** Execution times for various steps of the proposed hybrid scheduling technique, averaged over all task sets of a size
### Table 6
Energy consumption values predicted by the scheduler and measured by the runtime system, averaged values over 10 task sets each

| #Tasks | 10   | 20   | 40   | 80   | Total |
|--------|------|------|------|------|-------|
| Avg. predicted energy consumption (J) | 0.516 | 0.977 | 1.983 | 3.909 | 1.846 |
| Avg. real energy consumption (J)     | 0.504 | 0.961 | 1.965 | 3.865 | 1.824 |
| Avg. predicted versus avg. real energy consumption | 1.024 | 1.016 | 1.009 | 1.011 | 1.015 |
| Min. predicted versus real energy consumption | 0.993 | 1.009 | 0.986 | 0.991 | 0.986 |
| Max. predicted versus real energy consumption | 1.055 | 1.023 | 1.025 | 1.042 | 1.055 |

### 6.2 Experiments with a real streaming application

In addition to the experiments with synthetic task sets already presented in Reference 19, we have implemented a real streaming application, which was subsequently executed on a more advanced prototype of the runtime system detailed in Section 4. The application performs a basic edge detection on images. It consists of the following tasks:

1. READ IMAGE: read the image from a csv file,
2. ENHANCE CONTRAST: enhance the image’s contrast,
3. VSOBEL: apply a vertical sobel filter to the output of task 2,
4. HSOBEL: apply a horizontal sobel filter to the output of task 2,
5. COMBINE IMAGES: combine the output images of tasks 3 and 4,
6. WRITE IMAGE: write the resulting image to a csv file.

Task 2 features two different modes: it either simply checks whether contrast is sufficient (toggling mode if necessary) and forwards the input image as is via the buffer system, or it actually performs contrast enhancement (toggling mode if contrast is deemed high enough). A mode toggle influences the task’s behavior in the upcoming round. The tasks performing I/O, that is, tasks 1 and 6, must be executed sequentially, whereas the other four tasks can run in parallel (contrast enhancement only if it is actually performed). Figure 12 shows the application’s task graph.

As input data we have chosen 105 images of size 800 × 600 pixels from the Caltech Home Objects data set.20 The images were converted to grayscale and contrast was intentionally decreased to provoke occasional mode toggles of the contrast enhancement task. Figure 13 illustrates the individual steps of the transformation process.

In our experiments, we were primarily interested in whether the scheduler’s predictions regarding runtime and energy consumption are accurate, and especially whether the dynamic remapper’s overhead is acceptable, only this time in a more realistic scenario. In order to conduct experiments with the application sketched above we had to combine the runtime system, buffer management, dynamic scheduler/remapper, and application code into one comprehensive system. As a hardware platform, we stuck to the Intel Xeon E5-1620 v3 with four physical cores, 15 frequency levels (excluding turbo mode) and core-individual DVFS from the earlier experiments with synthetic task sets. To facilitate accurate scheduling predictions we again performed microbenchmarks with all tasks to determine core power consumption and runtime at each operating

---

**Figure 12** Task graph of the edge detection application
The tasks’ parallel efficiencies could be obtained as well. As it turned out, the runtime scales very well with the operating frequencies for all tasks considered here. Thus, the scheduler only needs a single value for each task to compute its runtime at a specific operating frequency. Table 7 gives an idea of the tasks’ relative sizes by providing the product of runtime (averaged over all inputs) and frequency, averaged over all frequencies. The I/O operations feature the longest runtimes, followed by the sobel filter application and contrast enhancement, if it is indeed performed. Table 7 also shows parallel efficiency values for all six tasks. The values reflect that the read and write task as well as the contrast enhancement task in check-only mode are executed sequentially. The sobel filter application and the task combining the two filtered images parallelize almost perfectly, while contrast enhancement still profits noticeably from parallelization.

In order to gather power consumption values for the scheduler, we initially took the same approach as for the experiments with synthetic tasks performing floating point operations. For each task and mode, power consumption was measured for each combination of core frequency levels, and the method of least squares was applied. Unfortunately, the least squares analysis yielded residuals roughly ten times as high for all tasks compared with the benchmark task from Section 6.1, and subsequent scheduling experiments with the values thus gained did not allow for accurate predictions of core energy consumption. Most likely, this is due to the workload being distributed statically and equally among all cores, which leads to some cores not running under load for the whole time when set to different frequency levels. We therefore employed a simpler technique to obtain the required values. For each task and frequency level, we measured power consumption when run on one, two, three, and four cores, subtracted idle power, divided by the number of utilized cores, and computed the average over these four values. That way, we received per-core power consumption values when executing a task at a given frequency and width with all involved cores constantly under load.

| Task                        | Avg. runtime (s) | frequency (MHz), averaged over all frequencies | Par. eff on #cores |
|-----------------------------|------------------|-----------------------------------------------|-------------------|
| READ IMAGE                  | 112.2            | 0.501 0.334 0.250                             |                   |
| ENHANCE CONTRAST (check-only) | 3.9              | 0.487 0.319 0.239                             |                   |
| ENHANCE CONTRAST (check and enhancement) | 35.3          | 0.893 0.807 0.737                             |                   |
| HSOBEL                      | 47.7             | 0.990 0.982 0.966                             |                   |
| VSOBEL                      | 50.1             | 0.987 0.980 0.970                             |                   |
| COMBINE IMAGES              | 23.3             | 0.990 0.978 0.969                             |                   |
| WRITE IMAGE                 | 123.5            | 0.500 0.333 0.250                             |                   |
In the following, we performed experiments with the edge detection application for six different deadline (i.e., round time) values. Deadlines were computed as

\[
M = d \cdot \frac{\sum_{j} t_j(f_{\text{max}})}{p} + \frac{\sum_{j} t_j(f_{\text{min}})}{p},
\]

where \(t_j(1, f)\) is the runtime of task \(j\) on a single core at frequency \(f\), and \(d\) is a parameter which allows us to vary the specified round length so as to cover the processor’s entire frequency range in our experiments. To this end, we set \(d \in \{0.7, 0.8, 1.0, 1.2, 1.4, 1.6\}\). We recorded the scheduler’s predictions on round execution time and energy consumption as well as the real values when running the application. Furthermore, remapper execution time was tracked and the schedules and energy value tables were stored for further analysis.

Table 8 presents the energy consumption values for each task and allocation at the respective energy-optimal frequencies for \(d = 1.0\). While the I/O tasks must be allocated an entire core each to keep the deadline, and the horizontal sobel filter task requires at least one core for half the round time, the other tasks can make do with any of the available allocations but at varying energy costs. Generally, higher values can be observed for small allocations (due to the higher operating frequencies needed to finish execution in time) and large allocations (due to the parallelization overhead). Table 9 shows the allocations the scheduler yields for \(d = 1.0\) and the two possible dynamic states of the application (contrast enhancement check-only or check and perform enhancement). Originally, the contrast enhancement task is allocated one core for a quarter of the round time, which is sufficient to execute the contrast check even at the lowest available frequency level. When it toggles its mode and thus contrast enhancement is carried out, the dynamic remapper extends the allocation for the contrast enhancement task to one core for half the round time, and operating frequency is set to the second lowest level. The allocations for the other tasks remain as they are since even after the mode toggle, the total processor allocation is at 3.75, that is, there are still resources left as higher resource utilization would not bring about a further reduction of energy consumption.

In the following, we focused our attention on the scheduler’s predictive capabilities for real applications. Table 10 provides information on per-round execution times and energy consumption for all examined deadline factors, both predicted by the scheduler and measured by the runtime system during execution of the application. The values are based on the execution rounds where each task could actually perform
worst predictions deviating ≈ 3% from reality. Energy consumption predictions cannot quite keep up here but are still accurate throughout, with average values better than 7% deviation for all deadline factors, and every single prediction within 10% of the experimentally measured values.

In order to demonstrate the effectiveness of our proposed framework, we have implemented a version of the edge detection application which does not make use of the framework. Each input image is fully processed within one round, and the tasks run one after another on all four available cores (reading and writing the image are still carried out sequentially). We have performed experiments with three different operating frequencies (1.2, 2.3, and 3.5GHz), and with a different scaling governor (ondemand), where frequency is not determined by the user but scaled dynamically according to the current load. Table 11 provides per-round execution times and energy consumption values. It becomes clear that both execution time and energy consumption increase when not employing the framework. The shortest execution times are achieved when setting all cores to the highest available operating frequency. Still, execution takes about 44% longer than with the framework for a deadline factor of 1.0, and energy consumption jumps to 277%. Execution takes slightly longer under the ondemand scaling governor but energy consumption decreases to 191%, the worst results regarding both energy consumption and round execution time are gained when running at the lowest available operating frequency.

The worst results regarding both energy consumption and round execution time are gained when running at the lowest available operating frequency.

Finally, we are interested in how long it takes to perform remapping at runtime when application behavior changes. For the current application, this is the case when the contrast enhancement task toggles its mode. Each time this happens, a dynamic remapping is invoked, and the next execution round commences with the schedule altered accordingly. It is imperative that remapping is performed as fast as possible, so the application will not be unduly stalled. Over the course of the application, that is, the processing of the 105 input images, 36 calls to the dynamic remapper were recorded. The average remapper execution time over the 196 total invocations (36 calls each in six experiments) was 4.6 μs with a worst-case value of 6.1 μs. Even at a round length of 38.9 ms (for the smallest deadline factor), the dynamic remapper’s execution time thus amounts to <0.016% of a round’s duration.

### Table 10

| Deadline factor | 0.7  | 0.8  | 1.0  | 1.2  | 1.4  | 1.6  |
|----------------|-----|-----|-----|-----|-----|-----|
| Avg. predicted round execution time (ms) | 38.606 | 44.121 | 53.713 | 66.025 | 74.829 | 88.242 |
| Avg. real round execution time (ms) | 38.914 | 44.566 | 54.351 | 65.625 | 73.637 | 88.866 |
| Min. pred. versus real round execution time | 0.977 | 0.972 | 0.972 | 0.991 | 1.003 | 0.978 |
| Avg. pred. versus real round execution time | 0.992 | 0.990 | 0.988 | 1.006 | 1.016 | 0.993 |
| Max. pred. versus real round execution time | 1.002 | 1.000 | 1.000 | 1.016 | 1.031 | 1.002 |
| Avg. predicted round energy consumption (J) | 1.261 | 1.245 | 1.293 | 1.378 | 1.463 | 1.548 |
| Avg. real round energy consumption (J) | 1.349 | 1.289 | 1.323 | 1.356 | 1.434 | 1.553 |
| Min. pred. versus real round energy consumption | 0.905 | 0.933 | 0.951 | 0.992 | 0.996 | 0.972 |
| Avg. pred. versus real round energy consumption | 0.935 | 0.965 | 0.978 | 1.016 | 1.020 | 0.997 |
| Max. pred. versus real round energy consumption | 1.013 | 1.032 | 1.037 | 1.073 | 1.084 | 1.049 |

### Table 11

| Operating governor | Ondemand | Userspace 1.2 | Userspace 2.3 | Userspace 3.5 |
|-------------------|----------|--------------|--------------|--------------|
| Avg. round execution time (ms) | 85.858 | 227.924 | 119.263 | 78.312 |
| Avg. round execution time versus framework (deadline factor 1.0) | 1.580 | 4.194 | 2.194 | 1.441 |
| Avg. round energy consumption (J) | 2.528 | 3.367 | 2.522 | 2.770 |
| Avg. round energy consumption versus framework (deadline factor 1.0) | 1.911 | 2.545 | 1.906 | 2.094 |
All in all, the experiments with the edge detection application largely confirm the high prediction accuracy the scheduler exhibited in the experiments with synthetic task sets. Furthermore, remapping can be achieved with very little runtime overhead, so the proposed approach is well suited for applications with a high degree of dynamic behavior.

7 Dynamic Scheduling for Heterogeneous Systems

So far, we have assumed a homogeneous platform, that is, that all processing cores are identical. We now assume a system with more than one type of cores, and multiple cores of each type. An example of such a system is ARM’s big.LITTLE architecture, for example, with four A7 cores and four A15 cores, which share a single memory address space and operating system instance.

A first, rather natural, restriction in the use of such a system is that a task, even if parallelized, will only use cores of one type. A second, possible restriction is if a task that runs on a first core type in one round, is allowed to run on a different core type in the next round. This might happen, for example, if some tasks that run on the first core type go from check-only mode to active mode, so that the cores of the first type are highly loaded and it would be more energy-efficient to run one of these tasks in the next round on a different core type, where the load is not as high. Balancing the load between core types allows for better choice of frequencies on all cores and thus for reduced energy consumption. Such a “migration” is possible if cores of both types can access the same (shared) memory, that is, can access the same buffers, and if the cores have a common ISA (as in big.LITTLE), or if specific code of this task for each core type (ISA) is available. Please note that core-type-specific code versions might also be useful in the case of identical ISAs, as micro-architecture oriented optimizations can be applied then.

If tasks cannot migrate to a different core type, then execution on a heterogeneous platform reduces to an initial partitioning of the tasks onto the different core types. Then, the mapping and remapping algorithm of Section 5 can be applied for each core type separately. The initial task partitioning can be obtained by finding an optimal mapping via an ILP as the discretized version of malleable task scheduling in Section 5 is a variant of a crown schedule, and crown scheduling has been extended to heterogeneous platforms. This, however, is only feasible if the application is run often enough or long enough to justify the energy investment into the optimization algorithm. Alternatively, a task partitioning can also be found by using a heuristic as described below.

If tasks are allowed to move to a different core type in the case of load imbalance to improve load balance and thus energy efficiency, then a quick decision is necessary which task should be moved, because this move will necessitate a remapping of tasks at runtime, that is, from one round to the next if possible. Hence, a balancing heuristic will be presented. Please note that this heuristic can also be used for the initial task partitioning: First, all tasks are assigned to one core type, and then the balancing is performed, that is, some task or tasks are moved to previously idle cores of different core type(s).

We first specify when the balancing heuristic will have to be invoked, that is, how we measure imbalance. In a scenario with malleable tasks, that is, with continuous allocation, a minimum energy is reached when the steepness of the energy curve is the same for all tasks. This property carries over to the heterogeneous case, even if the absolute energy values for the same workload on different core types may vary. In our discretized allocation case, we cannot achieve identical steepness values anymore, but the remapping heuristic still tries to keep the steepness values as close as possible. Hence, we compute the average steepness $\overline{st}_c$ over all tasks on each core type $c$ (possibly weighted with the relative workloads of the tasks), and if these values are farther apart than some predefined threshold $T$, then the balancing heuristic is invoked.

The balancing heuristic, which we present for the case of two core types, moves a task from the higher loaded cores to the cores of the other type. Then, the remapping heuristic is invoked for each core type, the average steepness computed for each core type, and compared against the threshold. The central part of the balancing heuristic is the choice of the task to be moved. Here, we apply a greedy approach: the task with the highest absolute value of steepness, that is, which is farthest away from the average steepness of the less loaded cores, is moved.

As an optimization, we can define a sequence $T_i$ of thresholds with $0 = T_0 < T = T_1 < T_2 < \ldots$ and move i tasks when

$$T_i < |\overline{st}_{c_1} - \overline{st}_{c_2}| < T_{i+1}.$$  

For $i = 0$, the balancing is not invoked. For $i \geq 2$, several tasks can be moved at once before executing the remapping heuristic again. The values of the $T_i$ will be derived from experiments.

We are aware that it is not the number of tasks alone that helps to balance the loads. Instead, the workloads of the tasks to be moved also play a role. However, as the large tasks are typically parallelized first, and hence their steepness values are lower than for small tasks, the chosen tasks, that is, those that are “lagging behind,” are normally smaller tasks.

8 Conclusion

We have extended the specification of parallel programs by streaming task graphs to express dynamic elements, that is, tasks that are not always executed. This helps to catch more application areas. As an initial step, we have restricted dynamic behavior to tasks that check at each invocation...
if they should toggle their state between active and check-only from the next invocation on. Much more general forms of dynamic behavior are possible, ranging from early termination of some tasks to advanced forms such as the possibility that a task can at runtime create another task and connect it in the task graph, which would in turn necessitate changes in the existing predecessor and successor tasks, at least with respect to adding another outgoing and ingoing buffer, respectively. This, however, might contradict the idea that the task graph structure is specified at high level, that is, not somewhere in the code of a task.

The experiments we have conducted show that the hybrid scheduler's energy consumption penalty compared with an optimal static scheduler is acceptable, while its massive runtime advantage enables deployment as a dynamic remapper. Further experiments indicate that its energy consumption predictions are fairly accurate. We have implemented a prototype system where users provide task code accessing communication buffers and specify the task graph structure, while the system compiles this into a running application by adding the communication buffer library and the runtime system. Experiments with a small real-world application on the runtime system indicate that the prediction of runtime and energy consumption done by the remapper is quite accurate, if the runtime and power profiles of the tasks are known. Moreover, the remapper produces a very low runtime overhead in this scenario. Finally, we have described how to extend our system to a heterogeneous platform like ARM’s big.LITTLE, as such platforms are getting more and more popular and have the potential to support energy efficiency further.

In the future, we would like to add more forms of dynamic behavior, such as concurrent tasks (with identical predecessors and successors) of which only one is active at any time. Furthermore, we would like to implement and evaluate the prototype system on a heterogeneous platform such as big.LITTLE, validate if the predictions of runtime and energy consumption are accurate also for other task types such as memory-bound tasks, as well as extend the energy consumption model from consumption by cores to consumption by caches, memories, and communication. Eventually, it would be interesting to explore the applicability of the dynamic remapper to systems with GPUs or other accelerators, as dynamic application behavior is conceivable in such an environment as well.

ACKNOWLEDGEMENT
Open access funding enabled and organized by Projekt DEAL.

DATA AVAILABILITY STATEMENT
The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID
Sebastian Litzinger © https://orcid.org/0000-0003-2200-7337
Jörg Keller © https://orcid.org/0000-0003-0303-6140

REFERENCES
1. Gordon MI, Thies W, Amarasinghe S. Exploiting coarse-grained task, data and pipeline parallelism in stream programs. In: Chen JP, Martonosi MR, eds. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII). New York, NY: Association for Computing Machinery; 2006:151-162.
2. Leung JYT, ed. Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Boca Raton, FL: CRC Press; 2004.
3. Sanders P, Speck J. Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks. In: Kaklamanis C, Papaioannou TS, Spirakis PG, eds. Proceedings of the 15th Conference on Algorithms and Architectures for Parallel Processing. Cham: Springer International Publishing; 2017:3-17.
4. Gill PE, Murray W, Wright MH. Practical Optimization. Bingley, UK: Emerald; 1982.
5. Litzinger S, Keller J. Influence of discretization of frequencies and processor allocation on static scheduling of parallelizable tasks with deadlines. In: Juurlink B, Karl W, eds. Proceedings of the 28th PARS-Workshop 2019. Berlin, Germany: Springer; 2020:95-107.
6. Marahatta A, Wang Y, Zhang F, Sangaiah AK, Tyagi SKS, Liu Z. Energy-aware fault-tolerant dynamic task scheduling scheme for virtualized cloud data centers. Mobile Netw Appl. 2019;24(3):1063-1077.
7. Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. ACM Comput Surv. 2005;37(4):316-344.
8. Keller J, Litzinger S, Spitzer W. Probabilistic runtime guarantees for statically scheduled taskgraphs with stochastic task runtimes. In: Lalande J-F, Moh T, eds. Proceedings of the 17th International Conference High Performance Computing & Simulation (HPCS). Cham: Springer; 2019.
9. Akkioka S. Task graphs of steam mining algorithms. In: Cormode G, Yi K, Deligiannakis A, Garofalakis MN, eds. Proceedings of the 1st International Workshop on Big Dynamic Distributed Data. Riva del Garda, Italy: Createspace Independent Publisher; 2013:55-60.
15. Falk J, Zebelein C, Haubelt C, Teich J. A rule-based quasi-static scheduling approach for static islands in dynamic dataflow graphs. ACM Trans Embed Comput Syst. 2013;12(3):74:1-74:31. https://doi.org/10.1145/2442116.2442124.

16. Holmbacka S, Müller R. epEBench: true energy benchmark. In: Merelli I, Lio P, Kotenko IV, eds. Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). IEEE; 2017:426-429. https://ieeexplore.ieee.org/document/7912683.

17. Kessler CW, Melot N, Eitschberger P, Keller J. Crown scheduling: energy-efficient resource allocation, mapping and discrete frequency scaling for collections of malleable streaming tasks. In: Henkel J, Yakovlev A, eds. Proceedings of the 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation. IEEE; 2013:215-222. https://ieeexplore.ieee.org/document/6662176.

18. Melot N, Kessler C, Eitschberger P, Keller J. Co-optimizing core allocation, mapping and dvfs in streaming programs with moldable tasks for energy efficient execution on manycore architectures. In: van der Aalst W, Keller J, Penczek W, eds. Proceedings of the 19th International Conference on Application of Concurrency to System Design (ACSD 2019). IEEE; 2019. https://ieeexplore.ieee.org/document/8843596.

19. Litzinger S, Keller J. Generating energy-efficient code for parallel applications specified by streaming task graphs with dynamic elements. PMAM '20 Association for Computing Machinery. New York, NY: Springer; 2020.

20. Caltech home objects dataset; 2006. Accessed Date April 24, 2020. http://www.vision.caltech.edu/pmoreels/Datasets/Home_Objects_06/.

21. Hähnel M, Härtig H. Heterogeneity by the numbers: a study of the ODROID XU+E big.LITTLE platform. In: Agarwal Y, Rajamani K, eds. Proceedings of the 6th Workshop on Power-Aware Computing and Systems, HotPower ’14. Broomfield, CO: USENIX Association; 2014.

22. Litzinger S, Keller J, Kessler C. Scheduling moldable parallel streaming tasks on heterogeneous platforms with frequency scaling. In: Greco MS, Pesavento M, eds. Proceedings of the 27th European Signal Processing Conference (EUSIPCO 2019). New York, NY: Springer; 2019.

How to cite this article: Litzinger S, Keller J. Code generation for energy-efficient execution of dynamic streaming task graphs on parallel and heterogeneous platforms. Concurrency Computat Pract Exper. 2022;34(2):e6072. https://doi.org/10.1002/cpe.6072