Design Automation for Embedded Systems, 4, 311–327 (1999)

Efficient Block Scheduling to Minimize Context Switching Time for Programmable Embedded Processors

INKI HONG
Computer Science Department, University of California, Los Angeles

MIODRAG POTKONJAK
Computer Science Department, University of California, Los Angeles

MARIOS PAPAEFTHYMIOU
EECS Department, University of Michigan, Ann Arbor

Abstract. Scheduling is one of the most often addressed optimization problems in DSP compilation, behavioral synthesis, and system-level synthesis research. With the rapid pace of changes in modern DSP applications requirements and implementation technologies, however, new types of scheduling challenges arise. This paper is concerned with the problem of scheduling blocks of computations in order to optimize the efficiency of their execution on programmable embedded systems under a realistic timing model of their processors. We describe an effective scheme for scheduling the blocks of any computation on a given system architecture and with a specified algorithm implementing each block. We also present algorithmic techniques for performing optimal block scheduling simultaneously with optimal architecture and algorithm selection. Our techniques address the block scheduling problem for both single- and multiple-processor system platforms and for a variety of optimization objectives including throughput, cost, and power dissipation. We demonstrate the practical effectiveness of our techniques on numerous designs and synthetic examples.

Keywords: Block scheduling, context switching minimization, architecture selection, algorithm selection.

1. Introduction

1.1. New Directions for an Ancient Problem

Scheduling has been one of the most popular research topics in DSP synthesis and compilation ever since the (r)evolution of implementation technologies enabled hardware sharing. Over the years, however, fine-grained scheduling of DSP operations lost some of its importance due to a variety of reasons. Extensive empirical studies have indicated that different competing scheduling techniques have relatively small impact on the key design metrics of the final implementation [11]. Recently developed estimation techniques have also indicated that there is little room for further improvement of scheduling algorithms [5, 32]. More importantly, there has been strong evidence that other behavioral synthesis tasks, in particular transformations, can have a significantly higher impact than scheduling on the quality of the final implementation [28].

The rapid pace of modern trends in DSP applications and implementation technologies has recently revitalized the importance of scheduling by simultaneously imposing new

sets of design issues, goals, and constraints. The current trend in DSP compilation and behavioral synthesis is to consider higher levels of abstraction and larger designs. This trend continues at increasing rate due to both application and implementation factors. The size of average DSP commercial electronics applications has been doubling each year. At the same time, the clock frequency and the average size of industrial integrated circuits have been doubling every three years. Thus, higher levels of hardware sharing become feasible and economically desirable with each new generation of technology, while at the same time hardware sharing is becoming increasingly important for satisfying applications requirements.

As we move toward systems-on-a-chip, DSP scheduling is inevitably changing its focus from operation-level granularity to task-level granularity. Thus, new scheduling problems arise that seek to optimize the efficiency of the execution of tasks (or blocks) using realistic timing models for the embedded processor platforms. Tasks may have task-level state variables shared among them. Such state can be any entity such as memory pointers, static variables, processor configuration settings, or a task's code that persists across different executions of a task and needs to be swapped out if a conflict arises with subsequently scheduled tasks. Due to this reason, different schedules of the tasks may result in different amount of context switching time. Therefore, at this level, context switching time is a new degree of freedom, and its minimization is a prime optimization objective.

In this paper, we present a collection of algorithmic techniques that generate optimal block schedules by minimizing the total context switch time. Our schemes yield optimal block schedules for any given platform and for any choice of algorithms for each task. Furthermore, our schemes can yield optimal block schedules while simultaneously enabling the selection of optimal architecture and algorithms for implementing the various tasks.

1.2. Block Scheduling Example

We introduce the block scheduling problem using the simple example described in Table 1. Given this table, we wish to find a schedule of blocks that maximizes throughput on a single-processor system. For the sake of simplicity, we assume that the iteration count for each block is 2, and that the blocks are independent. A block schedule $\langle A_1, B_1, C_1, D_1, E_1, A_2, B_2, C_2, D_2, E_2 \rangle$ takes 431 cycles. A different block schedule

Table 1. A simple instance of the block scheduling problem.

		Context Switching Times Between Blocks				
Block	Processing Time	A	B	C	D	E
A	30	0	18	20	15	10
B	15	8	0	15	8	7
C	20	12	8	0	22	15
D	10	10	10	18	0	32
E	25	19	8	12	10	0

$\langle A_1, A_2, B_1, B_2, C_1, C_2, D_1, D_2, E_1, E_2\rangle$, which has the property that different executions of each task are successively scheduled, takes 306 cycles. Another block schedule with the property, $\langle A_1, A_2, E_1, E_2, C_1, C_2, B_1, B_2, D_1, D_2\rangle$ takes 248 cycles, however. Since the order of execution does not affect processing times, only the switching times are to be considered during schedule optimization. Considering only the switching times, the three schedules takes 231, 103 and 48 cycles, respectively.

1.3. Paper Organization

The remainder of this paper has five sections. In Section 2 we present our computational and hardware model. We also briefly consider a simplified version of the general block scheduling problem in which no data dependencies are considered, and we argue that it is computationally intractable by a reduction from the well-known Traveling Salesman Problem (TSP). Section 3 discusses related research. In Section 4, we formulate, analyze, and solve the block scheduling problem with simultaneous architecture and algorithm selection and no data dependencies. Our results apply to both single- and multi-processor platforms. In Section 5, we address the general block scheduling problem with data dependencies on single-processor platforms. We summarize our results in Section 6.

2. Preliminaries

2.1. Computational and Hardware Model

This paper is concerned with the problem of scheduling computations in order to maximize the efficiency of their execution on programmable embedded systems. In this section we describe our computational model, our hardware model, and its associated cost functions.

We model a computation on a given system and a given algorithm implementation for each block as a directed edge-weighted, vertex-weighted graph $G = (V, E, d, p, s)$ which is called a *computation dataflow graph*. Each vertex $u \in V$ denotes a *computational block*, that is, a sequence of computations that are executed given an input. The edges in G encode *data dependencies*. Specifically, each edge $(u, v) \in E$ denotes that block v requires the output of block u in its computation. Note that the set E is only relevant in Section 5. We assume that the set E does not introduce cycles in the graph G. The number of iterations by which an output of block u must be delayed before it is presented to the inputs of block v is given by the integer *delay count* $d(u, v)$. Each vertex-weight $p(u)$ denotes the processing time for the block corresponding to vertex u. For each ordered pair of vertices $c(u, v)$ denotes the time required to switch processing from block u to block v. In this model, we assume that the benefit of state-sharing occurs only when the associated tasks are scheduled successively. The model has a limitation such that state-sharing between tasks not scheduled successively cannot be exploited.

The length $L(\mathcal{S})$ of a given schedule $\mathcal{S}$ for a single processor is given by adding up the total processing time $p(\mathcal{S})$ and the total switching time $s(\mathcal{S})$. Note that the schedule $\mathcal{S}$ is

an ordered set. The total processing time is given by the sum

$$p(\mathcal{S}) = \sum_{u \in \mathcal{S}} p(u),$$

which does not depend on the execution order of the blocks. The total switching time is given by

$$s(\mathcal{S}) = \sum_{\{u_i, v_j\} \subseteq \mathcal{S}} c(u, v),$$

where the subset notation denotes that the blocks u_i and v_j are executed consecutively in $\mathcal{S}$. Thus, the order in which blocks are scheduled to execute can affect the switching cost for the implementation of a computation. Context switching costs are assumed to be high, and thus no preemption is allowed. The length $L(\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m)$ of given schedules $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ for m processors, respectively is given by

$$L(\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m) = \max_{i=1,2,\ldots,m} L(\mathcal{S}_i).$$

In the block scheduling problem, given a computation G with an associated architecture an an algorithm for executing each block in G, we wish to find a schedule $\mathcal{S}$ of the blocks that satisfies all data dependencies in G and minimizes $L(\mathcal{S})$. The block scheduling problem can be augmented to include architecture and algorithm selection. In the augmented problem, we wish to find an optimal schedule for an optimal selection of architecture and algorithms from a specified set of alternatives. The augmented block scheduling problem may have several optimization objectives such as cost of components or power dissipation. For example, designers may have a choice among platforms with different voltage supply characteristics, and thus different power dissipation characteristics, assuming that power consumption is given by $V_{dd}^2 \cdot C_L \cdot \alpha \cdot f_{clock}$, where V_{dd} is supply voltage, C_L is switching capacitance, α is switching activity, and f_{clock} is a clock frequency.

2.2. Equivalence of Block Scheduling Problem and Traveling Salesman Problem

When no data dependencies exist between blocks, the block scheduling problem is equivalent to the Traveling Salesman Problem (TSP). The TSP is formally stated as follows.

Problem: Traveling Salesman Problem

Instance: Given is a set of n cities C, an $n \times n$ matrix S of distances between cities in C, and a positive number l.

Question: Is there a tour of C of length l or less, i.e., a permutation $(\pi(1), \pi(2), \ldots, \pi(n))$, $\pi(i) \in C$, such that $l \geq s_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} s_{\pi(i)\pi(i+1)}$?

When the matrix S is symmetric, i.e., $s_{ij} = s_{ji}$ for all i, j, then the TSP is called symmetric. When S is not symmetric, the TSP is referred to as asymmetric.

The correspondence between the simplified block scheduling problem above and the asymmetric TSP is straightforward. Each block u corresponds to a city, and the switching cost matrix S corresponds to the distance matrix. Thus, the optimum block schedule corresponds to a minimal length tour, and vice-versa. In Section 4 we consider the augmented block scheduling problem with simultaneous algorithm and architecture selection and no data dependencies. The equivalence of block scheduling and TSP is central to our attack to this problem.

3. Related Research

In this section we briefly summarize previous research on scheduling and on algorithm and architecture selection. We also outline the most relevant TSP references.

Scheduling has been widely studied in many areas including hardware/software cosynthesis [6], system-level synthesis [19], behavioral synthesis [7, 23], DSP [26, 33], parallel processing [10], and hard/soft real-time systems [21, 30]. The majority of the proposed DSP scheduling techniques has been presented in behavioral synthesis literature [7, 23]. Initial scheduling and transformation techniques for block processing are presented in [1, 3, 16, 34]. Task-level scheduling of a dataflow graph [18] has been actively researched. Researchers proposed algorithms to minimize the size and memory usage of the code generated from a dataflow graph specification [1, 2, 25]. Also, algorithm and architecture selection has been recently recognized as an important system-level synthesis topic [27].

The Traveling Salesman Problem is probably the most widely studied problem in combinatorial optimization mostly due to its wide applicability as well as its representative nature of difficult optimization problems [17]. TSP is known to be NP-complete [9]. A number of efficient and effective solution techniques have been presented for the symmetric TSP that come within a few percentage points from the optimum [12]. According to Johnson's comprehensive empirical comparison of symmetric TSP heuristics [12], the most efficient and effective heuristics are the 3-Opt and the Lin-Kernighan [20] algorithms. Reinelt [31] has effectively reduced the running times of these schemes by considering only $O(n)$ candidate edges for the tour. Martin *et al.* [22] have given an effective way to improve the quality of the solutions obtained by these heuristics at the expense of increasing their running times.

Although a few solution methods have been proposed for the asymmetric TSP, including a heuristic by Kanellakis and Papadimitriou [14] which modifies the Lin-Kernighan symmetric TSP heuristic and a branch-and-bound approach by Miller and Pekny [24], there do not exist enough experimental data to judge their performance, and the heuristics have very high time complexity.

4. Block Scheduling with Independent Blocks

In this section we describe our algorithms for the augmented block scheduling problem with independent blocks and simultaneous architecture and algorithm selection. We first outline our key underlying assumptions. We then formulate the augmented block scheduling problem for a variety of objectives, including throughput, cost, and power optimization.

Finally, we present effective TSP-based heuristics for solving these problems and present experimental results that support the effectiveness of the proposed schemes.

4.1. Problem Formulations

We assume that there are no data, control, or timing dependencies among blocks. In this case, only the blocks within a single iteration need to be scheduled, since any number of block iterations can be scheduled consecutively to generate the zero context switch time between the iterations of the block. For each block, we assume that there exist a number of algorithm choices. Also, there exist a number of processor types or implementation platforms. The switching time between the algorithms of any two blocks is considered as well as the running times for processing the blocks. Our schemes address two different design approaches: (i) only one processor is chosen for all blocks (ii) multiple processors (of possibly different types) are chosen, and each block is assigned to one them.

4.1.1. Throughput Optimization

The block scheduling problem for throughput optimization can be formally stated as follows.

Problem: Block Scheduling for Throughput Optimization with Simultaneous Algorithm and Architecture Selection

Instance: We are given n blocks, T different processor types, and a positive constant l. Each block i has a choice of B_i different algorithms. For each processor type, we are given the processing times array P for all the available algorithms and the switching times matrix S between the algorithms.

Question: Is there a schedule of blocks, a selection of algorithms and a selection of processors such that the resulting schedule length is at most l?

4.1.2. Cost Optimization under Throughput Requirement

The block scheduling problem for minimum cost with simultaneous selection of algorithms and architectures is similar to the throughput optimization problem. The only difference is that the objective is to minimize the cost under a throughput constraint.

Problem: Block Scheduling for Cost Optimization under Throughput Requirement with Simultaneous Algorithm and Architecture Selection.

Instance: We are given n blocks, T processor types, a cost array C for the processor types, and positive constants l and m. Each block i has a choice of B_i different algorithms. For each processor type, we are given the processing times array P for all the available algorithms, and the switching time matrix S between the algorithms.

Question: Is there a schedule of blocks, a selection of algorithms and a selection of processors such that the resulting schedule length is at most l, and the cost of all the processors selected is at most m?

4.1.3. Power Optimization under Throughput Requirement

Power optimization under throughput requirement with simultaneous algorithm and architecture selection and voltage scaling can be formulated in a way similar to throughput optimization. The differences are that the optimization objective is the minimization of the power consumption under the throughput requirement, and that voltage scaling is performed in addition to algorithm and architecture selection.

Problem: Block Scheduling for Power Optimization under Throughput Requirement with Simultaneous Algorithm and Architecture Selection and Voltage Scaling.

Instance: We are given n blocks, T different processor types with cost array C, a clock-cycle time array CCT, an array of power dissipations PD, an initial voltage V of each processor type, and positive constants l and m. Each block i has a choice of B_i different algorithms. For each processor type, we are given the running times array P for all the available algorithms and the switching times matrix S between the algorithms.

Question: Is there a schedule of blocks, a selection of algorithms, a selection of processors and a new voltage V' such that the resulting schedule length is at most l and the power consumption is at most m?

The power consumption is given by $P = \alpha C_L V_{dd}^2 f_{clock} = \alpha C_L V_{dd}^2 / CCT$, where CCT is a clock cycle time. It is known that reduced voltage operation comes at the cost of reduced throughput [4]. The gate delay GD follows the following formula: $GD = k \frac{V_{dd}}{(V_{dd} - V_t)^2}$ where k is a constant [4], and V_t is the threshold voltage. The maximum rate at which a circuit is clocked monotonically decreases as the voltage is reduced. The maximum clock cycle time $CCT(V')$ at a voltage level V' relative to the maximum clock cycle time $CCT(V)$ at a voltage level V is given by

$$CCT(V') = \frac{V'/(V' - V_t)^2}{V/(V - V_t)^2} \cdot CCT(V) = \frac{V'(V - V_t)^2}{V(V' - V_t)^2} \cdot CCT(V).$$

The power consumption $P(V')$ at a voltage level V' relative to the power consumption $P(V)$ at a voltage level V is given by

$$P(V') = \frac{V'(V' - V_t)^2}{V(V - V_t)^2} \cdot P(V).$$

4.2. TSP Implementation Issues

At the heart of all our schemes lies a block scheduler that is implemented efficiently using TSP heuristics. In this section we describe a few issues related to our implementations of

these heuristics. All our schemes transform an asymmetric TSP into a symmetric one using the technique proposed by Jonker and Volgenant [13]. We are thus able to apply directly efficient and effective heuristics for the symmetric TSP.

The transformation into symmetric TSP is performed as follows. Let TSP(C, S) denote an asymmetric TSP with a set of n cities $C = 1, 2, \ldots, n$ on a distance matrix S. Let TSP(C', S') denote the transformed symmetric TSP problem. Each city i in C is converted into two cities, i and $i + n$, in C'. If $|i - j| = n$, i.e., the two cities i and j in C' are from the same city in C, $s'_{ij} = -M$, where M is a very large positive number. If ($i \leq n$ and $j \leq n$) or ($i > n$ and $j > n$), $s'_{ij} = \infty$. Otherwise, $s'_{ij} = s_{ij}$ if $i > j$ and $s'_{ij} = s_{ji}$ if $i < j$. It is shown in [13] show that any optimal solution of the transformed symmetric TSP corresponds to an optimal solution of the original asymmetric TSP. Feuer and Koo [8] have successfully applied a similar transformation to the scan chain ordering problem.

Recent studies [12] suggest that the most efficient and effective heuristics for the symmetric TSP are 3-Opt and Lin-Kernighan [20] algorithms. These algorithms apply a greedy global optimization strategy which iteratively constructs an improved new solution s' from the previous solution s by making local changes to s. If no s' can be found, the solution s is returned as a local optimum. Martin *et al.* [22] have given an effective scheme, called the large-step Markov chain (LSMC), that improves the quality of these solutions. This heuristic is now considered to be the "champion" for symmetric TSP. LSMC starts with an arbitrary initial local minimum and keeps improving on it by "perturbation and local optimization" cycles. In our software implementation, we have used the LSMC heuristic with Reinelt's fast, nearest-neighbor based implementation of the 3-Opt local optimization engine [31]. The quality of the TSP tour can be traded off with running time by adjusting the number of the 3-Opt local optimization algorithms. In all our runs, we used the LSMC heuristic with 10 iterations of 3-Opt local optimization algorithms.

4.3. Block Scheduling on Single Processor

4.3.1. Algorithms

We have developed an iterative improvement algorithm based on the efficient implementation of a wide variety of TSP heuristic. Our throughput optimization heuristic is given by the pseudo-code in Figure 1. Our heuristic for cost optimization under a throughput constraint is described in Figure 2. Our heuristic for power optimization under a throughput constraint is given by the pseudo-code in Figure 3. In all experiments, the threshold t was chosen to be 100.

4.3.2. Experimental Results

We have constructed 4 different sets of 12 tasks from the 16 real-life tasks described in [29]. The tasks are the following: two GE controllers, two Honda controllers, a wavelet filter, four audio filters, a 8×8 discrete cosine transform, an NEC digital-to-analog converter, two components of modems, a LMS audio formatter, an echo canceler, and a linear controller

```
Set the schedule length of the best-so-far solution to ∞.
For each processor type,
  Generate an initial random solution.
  While number of consecutive iterations with no improvement < threshold t
    Apply TSP heuristic on current solution to get a block schedule.
    If there is improvement
        accept the new solution.
    Change current solution by changing algorithms of a randomly chosen block.
  Keep the current solution as the best solution for the processor.
  If best solution is better than best-so-far, set the best-so-far solution to the best solution.
Return the best-so-far solution.
```

Figure 1. Heuristic for throughput optimization.

```
Set the best-so-far solution to have the cost ∞.
For each processor type,
  Apply throughput optimization heuristic.
  If the schedule satisfies throughput requirement
    If the cost of processor type < the best-so-far cost
      Set the best-so-far solution to the current solution.
Return the best-so-far solution.
```

Figure 2. Heuristic for cost optimization under throughput constraint.

```
Set the power consumption of best-so-far solution to ∞.
For each processor type
  Generate a block schedule using throughput optimization heuristic.
  Scale the voltage so that the block schedule just satisfies the throughput constraint.
  Compute power consumption.
  If power consumption < best-so-far
    set best-so-far solution to the current solution.
Return the best-so-far solution.
```

Figure 3. Heuristic for power optimization under throughput constraint.

for automotive motion control. For each task, we have used the number of operations as the approximation of the running time. The context switching times are approximated such that they are proportional to memory requirement, the switching between similar tasks costs little overhead and the switching between different tasks requires a significant amount of overhead. The context switching costs are distributed between 10 and 173. We also

applied our algorithms on randomly generated computation graphs with 10, 25, 50, 75, and 100 blocks. We assumed 5 processor types with randomly chosen parameters and 5 algorithm choices for each block. Table 2(a) gives the processor specifications. For each processor type, processing and switching times were chosen in the range from 1 to 100, cost was chosen between 100 and 200, clock-cycle time was chosen in the range from 5.0 to 10.0, initial voltages were chosen between 3.0 and 5.0, and power dissipation was between 5.0 and 10.0. The threshold voltage was assumed to be 0.7. For each processor type, in addition to its specification, we give a lower bound on the length of any schedule with a single iteration. This bound is derived by summing up the smallest processing times of the blocks and the smallest switching times of all block pairs.

Table 2(b)–(d) illustrates the effectiveness of our TSP-based heuristics on the 50-block computation, a representative example among the ones we experimented with. We compared the effectiveness of our algorithms with the best among 100 random solutions that were generated for the problem. For this specific example, the average improvement for throughput and power was a factor of 2.9 and 2.4, respectively. For cost optimization under throughput requirement, no random solution satisfied the upper bound on the schedule length.

Tables 3 and 4 summarize the experimental results for all real-life design examples and all randomly generated computations, respectively. For each block count, we generated 10 random computations.

4.4. Block Scheduling on Multiple Processors

4.4.1. Algorithms

We developed an exhaustive-enumeration algorithm for cost optimization under throughput requirement that was based on TSP heuristics and lower-bound estimations. Similarly to the lower-bound estimation technique for block scheduling on a single processor, we designed a simple and effective lower-bound estimation technique for throughput optimization on multiple processors. For each block, we choose an algorithm with the smallest sum of processing and switching times over all processor types. A lower bound on the length of a schedule for the multiple-processor block scheduling problem can be obtained by dividing the sum of the processing and switching times for the best algorithms by the number of processors. Using this lower-bound, we can prune infeasible branches of the search. The heuristics for throughput optimization and for cost optimization under throughput requirement are given in Figures 5 and 4, respectively. The threshold t was chosen to be 10, and the value of k was set to 5 to allow escaping from local minima. These values were determined based on empirical observation obtained from extensive experimentations.

4.4.2. Experimental Results

We generated random computations with 10, 25, 50, 75, and 100 blocks. For each block count, we generated 10 computations graphs. As in the single-processor experiments, we assumed 5 processor types and 5 algorithm choices for each block. Running and switching

Table 2. Experimental results for a computation with 50 blocks on a single processor. In (a), LB stands for lower-bound. In (c), no random solution satisfied maximum schedule length constraint. In (d), sample period was 50,000 time units.

	Processor Type				
	#1	#2	#3	#4	#5
Cost	171	126	165	119	155
Clock-cycle time	9.1	10.0	7.1	7.4	5.9
Initial voltage	4.2	5.0	4.3	4.3	4.2
Power dissipation	8.4	6.9	6.4	9.7	5.8
LB on schedule length	1701	1481	1311	1480	1657

(a) Processor specifications.

	Random Solution	Heuristic Solution
Processor Type	#2	#3
Schedule length	4304	1476

(b) Throughput optimization results.

Requirement	1900	1800	1700	1600	1500
Schedule length	1653	1653	1653	1476	1476
Cost	119	119	119	165	165
Processor type	#4	#4	#4	#3	#3

(c) Cost optimization under throughput requirement.

	Random Solution	Heuristic Solution
Processor type	#5	#3
Scaled clock-cycle time	11.09	33.88
Initial voltage	4.2	4.3
Scaled voltage	2.77	1.75
Scaled power dissipation	2.53	1.06
Total energy consumption	126590.44	53124.29
Schedule length	4508	1476

(d) Power optimization under throughput requirement.

Table 3. Average improvement factors for synthetic examples on a single processor.

Blocks	Throughput Improvement	Cost Improvement	Power Improvement
10	2.8	2.9	2.1
25	3.1	3.5	2.4
50	2.9	3.4	2.3
75	2.7	2.9	2.2
100	3.2	3.7	2.6

Table 4. Average improvement factors for real-life designs on a single processor.

	Throughput Improvement	Cost Improvement	Power Improvement
Set 1	2.1	2.3	1.8
Set 2	2.0	2.2	1.7
Set 3	1.9	2.1	1.7
Set 4	2.2	2.5	1.9

Randomly assign the same number of blocks to each of the allocated processors.
For each processor in the processor set,
  Apply the heuristic for throughput optimization on the processor to get a block schedule.
While the number of consecutive iterations with no improvement $<$ a threshold t
  Move a randomly chosen block from the processor with the maximum schedule length to another processor which is chosen by the proportional probability depending on the schedule length.
  If the resulting maximum schedule length is not larger than k% more of the best-so-far maximum schedule length
    the current assignment of blocks is updated with the change.
  If it is smaller than the best-so-far schedule and assignment
    the solution is updated.
Return the best-so-far solution.

Figure 4. Heuristic for throughput optimization on multiple processors.

The current processor set is empty.
Repeat
  Choose the next cheapest processor set as the new current processor set.
  Compute the lower-bound for the current processor set.
  If the lower-bound satisfies the throughput requirement
    apply the heuristic to optimize throughput on multiple processors in Figure 4.
Until the current processor set satisfies the throughput requirement.
Return the current processor set and block schedules.

Figure 5. Heuristic for cost optimization under throughput requirement on multiprocessors.

times were randomly chosen in the range from 1 to 100, and the costs of the processors were chosen between 100 and 200. Table 5 gives cost optimization results under a throughput requirement for the example computation with 50 blocks. The results for all examples are illustrated in Table 6. Random solutions were generated by picking the best among 100 random schedules of blocks to processors which were allocated by exhaustively enumerating all allocation possibilities in increasing cost order. Under various throughput requirements, the average improvement obtained over all examples was a factor of 3.1.

Table 5. Results of cost minimization for 50-blocks computation on multiple processors: IF – improvement factor.

	Cost			Schedule Length	
Sample Period	Random	Opt	IF	Random	Opt
500	1408	416	3.38	458	486
600	1040	312	3.33	577	595
750	766	208	3.68	738	749
850	684	208	3.29	845	749
1000	520	208	2.5	970	749
1500	312	104	3.0	1472	1389

Table 6. Average cost improvement factors on multiple processors over all examples generated.

Blocks	Cost Improvement
10	2.6
25	2.8
50	3.3
75	3.1
100	3.5

5. Block Scheduling with Dependent Blocks

In this section we address the block scheduling problem for throughput optimization with data dependencies. We assume that the given computation dataflow graph is acyclic. As in the case for independent blocks, only the blocks within a single iteration need to be scheduled, since any number of block iterations can be scheduled consecutively to generate the zero context switch time between the iterations of the block. However, as a preprocessing step, we remove all the dependencies associated with edges that have delay greater than or equal to the iteration count. We focus on single processor platforms without considering algorithm and architecture selection.

5.1. Problem Formulation

The block scheduling problem for throughput optimization can be formally stated as follows.

Problem: Throughput Optimization

Instance: We are given an iteration count I, n blocks with dependencies described by a computation dataflow graph G, the processing times array P for all the blocks, the context switching times matrix C between the blocks, and a positive constant l.

Question: Is there a schedule of blocks such that data dependencies are satisfied and the schedule length is at most l?

The throughput optimization problem is clearly intractable, since its special case with no dependency is already intractable. We assume that the given computation dataflow graph is acyclic. As in the case for independent blocks, only the blocks within a single iteration need to be scheduled, since any number of block iterations can be scheduled consecutively to generate the zero context switch time between the iterations of the block. However, as a preprocessing step, we remove all the dependencies associated with edges that have delay greater than or equal to the iteration count.

5.2. Verifying Valid Schedules

In this subsection we show how to verify whether a schedule is *valid*, that is, whether it correctly executes a given computation. We first give two necessary and sufficient conditions for a schedule to be valid. We then describe a simple scheme that checks these conditions.

Given a computation $G = (V, E, d, p, s)$ and schedule $\mathcal{S}$, the following two conditions are necessary and sufficient for $\mathcal{S}$ to be valid.

(C1) For every computational block $u \in V$, we must have that u is in the schedule.

(C2) For every pair u, v in V such that there exists a path from u to v in G such that $d(u, v) \leq I$, we must have that u precedes v in the schedule.

The first condition ensures that every computational block computes an output. The second condition ensures that data dependencies are not violated.

If the schedule $\mathcal{S}$ is finite, conditions C1 and C2 can be verified in $O(|\mathcal{S}|^2)$ time, where $|\mathcal{S}|$ is the number of blocks in the schedule. Note that $|\mathcal{S}| = |V|$. Verifying condition C1 requires a single scan of the schedule, which can be performed in linear time. In order to verify condition C2, we first compute the transitive closure on G to determine reachability. When computing the transitive closure, we compute the delay values of all edges and remove the edges that have delay greater than or equal to the iteration count. We then perform a pairwise check of all blocks in the schedule to determine whether the precedence constraint C2 is satisfied between blocks that are reachable.

5.3. Optimization Algorithm

For the solution of the throughput optimization problem, we have applied a general combinatorial optimization technique known as simulated annealing (SA) [15]. The cost function is the length of the schedule. The initial schedule presented to the SA algorithm is obtained by a topological sort of the computation dataflow graph. Subsequent solutions are generated by randomly swapping two blocks in the schedule until we obtain a valid new schedule. We randomly choose a pair of blocks to swap in a schedule. For all the blocks between the pair in the schedule, we need to check if they violate any dependency relationship with the pair. This check can be efficiently performed using the transitive closure of the dataflow

Table 7. Results of throughput optimization with data dependencies: N_B – number of blocks, RS – random solution, HS – heuristic solution, AIF – average improvement factor.

	Iteration Count					
	2		3		5	
N_B	RS	HS	RS	HS	RS	HS
5	529	292	696	394	982	577
8	824	439	1138	633	1489	851
10	1056	569	1388	795	1991	1150
15	1521	800	1912	1062	2844	1663
20	2174	1187	2731	1566	3993	2264
AIF	—	1.9	—	1.8	—	1.7

computation graph. To reduce the running time, we have used a window parameter w such that only a pair of blocks within w blocks away in the schedule can be selected. As we use larger w, we get better results with larger running time. We repeat the process until we get a valid schedule. At any time during the execution of the algorithm, only valid schedules are allowed. SA escapes local minima by allowing worse solutions to be accepted as new starting solutions with an initially high probability that gradually decreases as the optimization process continues.

5.4. Experimental Results

We randomly generated computations with 5, 8, 10, 15, and 20 blocks. We considered iteration counts of 2, 3, and 5. The number of edges in the computation dataflow graph for a single iteration was $\frac{\#\ blocks}{2}$, and the number of edges with a single delay was $\frac{\#\ blocks}{5}$. The processing times and switching times were randomly chosen in the range between 1 and 100. The results of our experiments are given in Table 7. The best of 100 random solutions are reported and compared with the heuristic solution. Observe that the improvement factor decreases as the number of iteration count increases, since the percentage of the processing time over the total execution time becomes higher.

6. Conclusion

This paper addresses the problem of scheduling blocks of computations in order to optimize the efficiency of their execution on programmable embedded systems under a realistic timing model of the processor. We leveraged on our approach to the block scheduling problem to address simultaneous algorithm and architecture selection for both single and multiple processor implementation platforms. The optimization objectives we considered were throughput, cost, and power. We demonstrated the effectiveness of our schemes on numerous examples. The target of our approach is on embedded software, but our results

can be applied whenever static scheduling is used to map task graphs onto any type of dynamically reconfigurable hardware.

Notes

In this paper, we use the term "block scheduling" to refer to task scheduling to minimize the context switch time. Block scheduling is distinct from "block processing", which performs vectorization of tasks.

References

1. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Optimal parenthesization of lexical orderings for DSP block diagrams. VLSI Signal Processing Workshop, pp. 177–186, 1995.
2. S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 42(3): 138–50, 1995.
3. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Design Automation for Embedded Systems* 2(1): 33–60, 1997.
4. A. P. Chandrakasan, S. Sheng, and R. W. Broderson. Low power CMOS digital design. *IEEE Journal of Solid-State Circuits* 27(4): 473–484, 1992.
5. S. Chaudhuri and R. A. Walker. Computing lower bounds on functional units before scheduling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4(2): 273-9, 1996.
6. P. Chou and G. Borriello. Software scheduling in the co-synthesis of reactive real-time systems. Design Automation Conference, pp. 1-4, 1994.
7. G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, NY, 1994.
8. M. Feuer and C. C. Koo. Method for rechaining shift register latches which contain more than one physical book. *IBM Technical Disclosure Bulletin* 25(9): 4818–4820, 1983.
9. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.
10. M. J. Gonzales. Deterministic processor scheduling. *ACM Computing Surveys* 9(3): 173–204, 1977.
11. R. Jain, A. Mujumdar, and A. Sharma. Empirical evaluation of some high-level synthesis scheduling heuristics. Design Automation Conference, pp. 686–689, 1991.
12. D. S. Johnson. Local optimization and the traveling salesman problem. In *Proceedings of the 17th Intl. Colloquium on Automata, Languages and Programming*, pp. 446–460, 1990.
13. R. Jonker and T. Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters* 2(4), 1983.
14. P.-C. Kanellakis and C. H. Papadimitriou. Local search for asymmetric traveling salesman problem. *Operations Research* 28: 1086–1099, 1980.
15. S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science* 220(4598): 671–680, 1983.
16. K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak. Optimizing systems for effective block-processing: The *k*-delay problem. Design Automation Conference, pp. 714–719, 1996.
17. E. L. Lawler, J. K. Lenstra, A. Rinnooy-Kan, and D. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, Chichester, 1985.
18. E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE* 83(5): 773–801, 1995.
19. C. Lee, M. Potkonjak, and W. Wolf. System-level synthesis of application specific systems using A* search and generalized force-directed heuristics. International Symposium on System Synthesis, pp. 2–7, 1996.
20. S. Lin and B. W. Kernighan. An effective heuristics algorithm for the traveling-salesman problem. *Operations Research* 31: 498–516, 1973.

21. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM* 20(1): 46–61, 1973.
22. O. Martin, S. W. Otto, and E. W. Felten. Large-step Markov chains for the TSP incorporating local search heuristics. *Operations Research Letters* 11(4): 219–224, 1992.
23. M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE* 78(2): 301–317, 1990.
24. D. L. Miller and J. K. Pekny. Exact solution of large asymmetric traveling salesman problems. *Science* 251: 754–761, 1991.
25. P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design* 11(1): 41–70, 1997.
26. P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on CAD* 8(6): 661–679, 1989.
27. M. Potkonjak and J. Rabaey. Algorithm selection: A quantitative computation-intensive optimization approach. International Conference on Computer-Aided Design, pp. 90–95, 1994.
28. M. Potkonjak and J. Rabaey. Optimizing resource utilization using transformations. *IEEE Transformations on CAD* 13(3): 277–292, 1994,
29. M. Potkonjak and W. Wolf. Cost optimization in ASIC implementation of periodic hard-real time systems using behavioral synthesis techniques. International Conference on Computer-Aided Design, pp. 446–451, 1995.
30. K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE* 82(1): 55–67, 1994.
31. G. Reinelt. Fast heuristics for large geometric traveling salesman problems. *ORSA Journal on Computing* 4(2): 206–217, 1992.
32. A. Sharma and R. Jain. Estimating architectural resources and performance for high-level synthesis applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 1(2): 175–90, 1993.
33. W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, J. L. van Meergergen, and A. van der Werf. Improved force-directed scheduling in high-throughput digital signal processing. *IEEE Transactions on CAD* 14(8): 945–960, 1995.
34. V. Zivojnovic, S. Ritz, and H. Meyr. Retiming of DSP programs for optimum vectorization. *International Conference on Acoustic, Speech, and Signal Processing* 2: 465–468, 1994.