Exact and Approximate Task Assignment Algorithms for Pipelined Software Synthesis

Matin Hashemi, Soheil Ghiasi
Electrical and Computer Engineering
University of California, Davis
CA 95616, USA

{hashemi,ghiasi}@ucdavis.edu

ABSTRACT

Pipelined execution of streaming applications enable processing of high-throughput data under performance constraint. We present an integrated approach to synthesizing pipelined software for dual-core architectures. We target streaming applications modeled as task graphs that are amenable to static analysis. We develop a versatile task assignment algorithm that considers the combined effect of workload imbalance between processors and inter-processor communication. Our technique, which runs in pseudo-linear time, provably maximizes application throughput. Furthermore, we develop an approximation algorithm for task assignment whose complexity is strictly polynomial. It provides the designer with an adjustable knob to controllably trade solution quality with algorithm runtime and memory requirement. Empirical throughput measurements using an FPGA-based dual-core system validate our theoretical results. Our exact algorithm consistently outperforms a recent competitor. Compared to exact task assignment, the approximate method runs about 3 times faster, requires about 20 times less memory, and results in only 1% to 5% throughput loss.

1. INTRODUCTION

Significant number of embedded applications are characterized by their requirement to process virtually-infinite flow of input data under performance constraints [2]. Such applications, generally referred to as *streaming applications*, appear in many disciplines such as networking, signal processing, security, and multimedia. Typically, streaming applications demand high throughput, but are not very sensitive to response latency. Thus, pipelined execution is a favorable design choice for their implementation [5].

On the other hand, continuation of Moore's law has enabled economical integration of many transistors on the same die. This capability, along with fundamental limitations of instruction-level parallelism and energy inefficiency of single-core performance scaling have led to emergence of multicore architectures [9]. Various manufacturer and research groups have demonstrated effectiveness of multicore processors in both general purpose and embedded computing [6, 10]. Multicores provide promising platforms for exploiting inherent vertical parallelism of streaming applications.

The process of application software development for parallel architectures is fairly unproductive and challenging today [1]. As a step in addressing this issue, we present a methodology for synthesizing pipelined streaming applications that execute on distributed-memory dual-core processors. Our compilation framework admits applications modeled as task graphs or acyclic synchronous dataflow [8]. Following static task scheduling [7], they are assigned to processors in the system, and executable codes are generated.

In order to maximize application throughput, we develop a theoretically optimal task assignment algorithm that jointly considers both inter-processor workload imbalance and communication. The technique is versatile in that it can provably optimize a reasonably-arbitrary function of the two factors. Our algorithm runs in pseudo-linear time with respect to problem size, and hence, its runtime also depends on workload intensity of application tasks. We extend our technique to a strictly-polynomial approximation of optimal task assignment. The approximation algorithm takes as input a *tolerable error bound*, and guarantees that solution quality is not degraded beyond the bound. The algorithm runtime and memory requirement are improved with loosening of the error bound. Thus, it serves as an *adjustable knob* for trading application throughput with compilation runtime and memory requirement.

In order to validate practicality of our theoretical contributions, we prototype the architectures on an FPGA board. Compared to a recent compilation scheme, our exact task assignment technique improves application throughput 6.4%. Furthermore, our approximation method offers a range of runtime-throughput tradeoff points. For example, it runs about 3 times faster, requires about 20 times less memory, and results in throughput degradation of about 1% to 5%.

2. BACKGROUND AND PRELIMINARIES

Application Model: We adopt the commonly used task graph model to represent the computation and communication involved in an application. An application is represented as a directed-acyclic graph (DAG) G in which, vertices of G denote the tasks or constituting computations of the application, and edges of G represent inter-task communication. Each task can be fired, i.e. executed on a processing resource, if all of its input data are available. Application inputs are ready at the beginning of analysis.

Note that several other models of computation can be converted to task graphs. For example, acyclic synchronous dataflow graphs are equivalent to task graphs following static scheduling [8]. Hence, our framework and results are readily extensible to such models of computation. We use w_v to refer to estimated computation workload of task v in cycles. The terms computation workload, latency, and delay of a node are used interchangeably throughput the paper. Also, let $c_{uv} = c(uv)$ denote the amount of data that needs to be communicated from task u to task v over edge e_{uv}.

Abstract Hardware Model: Given task graph representation of a streaming application, we aim to realize the application by synthesizing two parallel pieces of software that execute on embedded dual-core architectures. Dual processing cores enable pipelined execution of the application, which in turn, improves application throughput. Figure 1 illustrates an abstract view of our target hardware architecture. The hardware is comprised of two simple processors (in-order issue) that are connected using a FIFO channel.

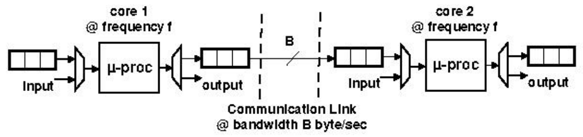


Figure 1: Target hardware architecture. Processors communicate using FIFO links.

We assume that processing cores are simple embedded processors that operate at the same frequency f. Inter-processor communication link provides bandwidth of B byte per second. Thus, communicating b bytes of data from one processor to another takes b/B seconds. In comparison, inter-task communication latency is negligible when the tasks are mapped to the same processor. Another way to view this is to incorporate intra-processor inter-task communication latency into tasks estimated workload (w_v).

Our objective is to develop a task graph partitioning and task assignment algorithm that judiciously favors computation workload or communication cost for given values of f and B. In the general case, such a versatile technique would be superior to conventional workload-balancing or communication minimization approaches.

Execution and Performance Model: We focus on pipelined execution of an application on target hardware architectures. We consider partitioning and assignment of application tasks onto pipeline processors. Figure 2.a shows an example in which, cut 1 partitions the graph into two sections. Tasks in the top section, i.e., a and b are assigned to core 1, and tasks in the bottom partition (c and d) will be executed on core 2.

Pipelined execution model implies that tasks assigned to different processors should not cyclically depend on each other [3]. Cycles in data dependency entangle task schedules on the two processors, which leads to degradation of pipeline throughput. Cut 2 in Figure 2.a illustrates an example.

Definition 2.1. *A partitioning is* convex *if and only if there is no cycle in dependencies among tasks in different partitions. In other words, the flow of data between convex partitions is uni-directional.*

Only *convex* partitions of task graph lead to competitive solutions and hence, are valid partitions for our purpose. For example in Figure 2.a, cut 1 is convex, but cut 2 is non-convex.

In the steady state, performance is determined by the slowest of 1) computation latency of any processing core, or 2) communication latency of the inter-processor link. In dual-cores, throughput is determined by W_{G_1}, C_C and W_{G_2}, where G_1 and G_2 denote the two partitions of graph G, W_{G_1} and W_{G_2} is the workload of the first and second partition,

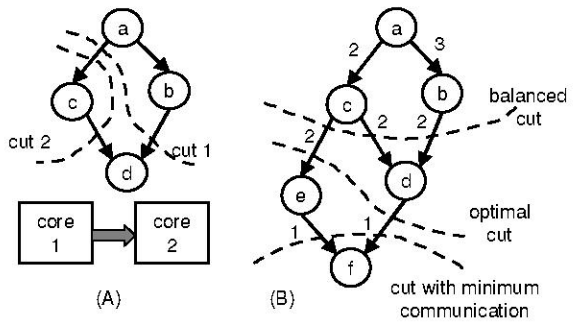


Figure 2: a) Pipelined execution of an example application. Cut 1 is convex, while cut 2 is not. b) Holistic workload balancing and communication minimization improves throughput.

and C_C denotes communication between the two partitions, i.e., cost of cut C. Let Q_C denote a hardware-driven function that estimates throughput for a given task assignment, i.e., a given cut C. An intuitive throughput estimation function would be $Q_C = max(\frac{W_{G_1}}{f}, \frac{C_C}{B}, \frac{W_{G_2}}{f})$.

In the remainder of this paper, we assume that task workload and inter-task communication are already converted to *latency*, unless otherwise noted. That is, we will assume that w_v is the workload of task v in time (considering f), and c_{uv} is communication latency of e_{uv} in time (considering B) on a particular architecture. Therefore, $Q_C = max(W_{G_1}, C_C, W_{G_2}) = max(W_{G_1}, C_C, W_G - W_{G_1})$, where W_G denotes workload of the entire application.

3. OPTIMAL TASK ASSIGNMENT

In this section, we present provably-optimal throughput-driven application partitioning, TAP, that maximizes throughput by minimizing Q_C. The method works on planar graphs, and thus, if the input SDF graph is not planar it goes through a planarization phase which is discussed in Section 5. That section also shows that a large class of streaming applications are intrinsically planar.

Let $G(s,t)$ denote the planar application task graph with source vertex s and sink vertex t (Figure 3.a). Recall that G is planar if and only if it can be drawn on a two-dimensional (2D) plane with no crossing edges. Given a planar DAG G, its *dual* graph G^* is well-defined by assigning vertices to faces of G, and connecting them using dual edges. The dual graph of a planar DAG $G(s,t)$ is denoted by $G^*(\nu_s, \nu_t)$ (Figure 3.b).

We follow a number of steps to find the path in G^* that minimizes Q_C. First, we move node weight values (task workloads) from vertices to edges in the following manner: Each node $v \in V(G)$ propagates the sum of its weight $w(v)$ and the propagated weights from its incoming edges to its right-most outgoing edge. Figure 3.a shows an example. Therefore, there are two values associated with each edge. For example in Figure 3.a, $edge(a,c)$ has two values: cost $c = c(a,c)$, and also a weight $w = 0$. Moving weights in this manner has the following interesting property:

Theorem 3.1. *For every convex cut C in G, W_{G_1} is equal to W_C. (note that $W_{G_1} = \sum_{v \in G_1} w(v)$ and $W_C = \sum_{e \in C} w(e)$).*

Figure 3.a shows an example. Sum of the weights of cut edges (dotted lines) is $W_C = (0) + (w_e) + (w_c) + (w_a + w_b)$,

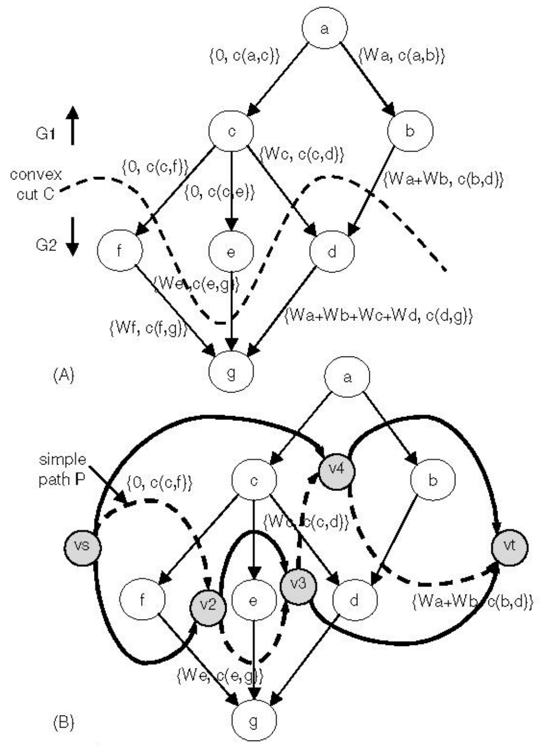


Figure 3: A) moving weights from vertices to edges. B) constructing the dual graph. A convex cut C and its corresponding simple path P are illustrated.

which is equal to sum of the weights of all vertices in the upper partition $W_{G_1} = w_a + w_b + w_c + w_e$. In general, the above theorem states that we can obtain both the computation workload of the upper-partition of the cut C and the inter-partition communication traffic, by only traversing cut C. This means that the cost function Q_C can be calculated as $Q_C = max(W_C, C_C, W_G - W_C)$ which depends only on edges in cut C and no other parts of the graph.

Subsequently, we construct the dual graph $G^* = (V^*, E^*)$ from the original task graph $G = (V, E)$. We transfer $w(e)$ and $c(e)$ values to the corresponding edge (e^*) in G^*:

$$\forall\ e \in E\ \exists\ \epsilon \in E^*$$
$$c(\epsilon) = c(e) \text{ and } w(\epsilon) = w(e)$$

Observe that a convex cut C in $G(s,t)$ is a simple path P from ν_s to ν_t in G^* (Figure 3.b). It implies that s and t belong to different partitions, and any path in G from s to t is cut exactly once.

$$Q_C = Q_P = max(W_P, C_P, W_G - W_P)$$

C_P and W_P are simply C_C and W_C but are calculated from the path P which is dual of the cut C. $W_C = W_P = \sum_{\epsilon \in P} w(\epsilon)$ and $C_C = C_P = \sum_{\epsilon \in P} c(\epsilon)$.

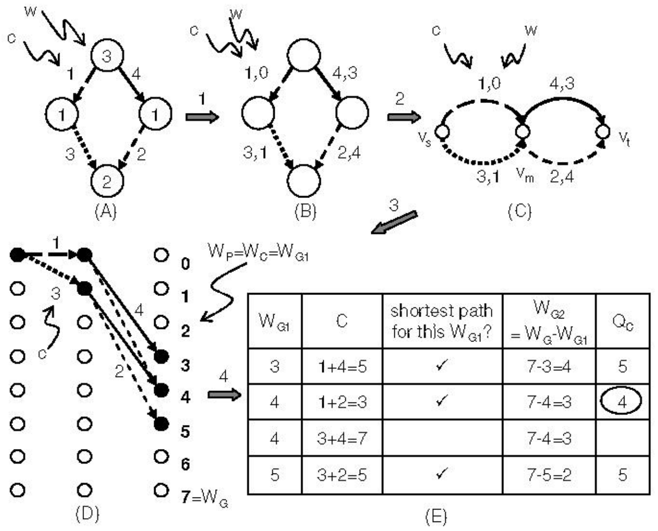
Although Q_C can be readily calculated using cut C, there could be exponentially many convex cuts in the graph. We argue that not all such cuts have to be evaluated. Let $G' = (V', E')$ denote the expanded G^* that is created according to the following rules:

$$V' = \{\nu_w : \nu \in V^*, 0 \le w \le W_G\}$$
$$E' = \left\{(\nu_{i_w}, \nu_{j_{w+w(\nu_i,\nu_j)}}) : (\nu_i, \nu_j) \in E^*\right\}$$

That is, G' is constructed from G^* by duplicating each vertex W_G times. Edges in G' have no weights, and their cost is equal to the cost of corresponding edge in G^*. For every edge (ν_i, ν_j) in G^* there are W_G corresponding edges in G', from vertex ν_{i_w} to vertex $\nu_{j_{w+w(\nu_i,\nu_j)}}$ for all $0 \le w \le W_G$, where $w(\nu_i, \nu_j)$ is weight of the edge (ν_i, ν_j) in G^*. However, if we fix the starting vertex, let's say $\nu_{i_{w=x}}$, then there is obviously one corresponding edge in G' which is $\nu_{j_{x+w(\nu_i,\nu_j)}}$.

Therefore, for every path P in G^* from ν_s to ν_t, there is one and only one corresponding path P' in G' if we start P' from a fixed $\nu_{s_{w=x}}$. If we fix $x = 0$ and always start from ν_{s_0}, then P' will end in ν_{t_w} where w is equal to W_P, because by definition W_P is sum of the above mentioned $w(\nu_i, \nu_j)$ for all edges in the path. Therefore, *index* of the vertex where P' ends in G' is equal to W_P, or equivalently gives the summation of task workloads in the top partition (Figure 4).

If two paths from ν_{s_0} arrive at the same ν_{t_m}, they both incur the same workload of m in the top partition. Therefore, only the path with minimum cost needs to be considered. Hence, we run the single-source shortest path on graph G', with one source ν_{s_0} and multiple destinations $\nu_{t_w} (\forall 0 \le w \le W_G)$. This will give us the minimum *cost* path among all paths with the same weight in G^*. In other words, this will give us one minimum-cost path P for every weight amount W_P from 0 to W_G. For every weight amount ranging from 0 to W_G, since $Q_P = max(W_P, C_P, W_G - W_P)$ and C_P is minimum, we have the minimum Q for that weight amount. Finally, we search among all W_G number of minimum values of Q and pick the one which is globally minimum (Figure 4.e). The expanded graph G' of the previous example in Figure 3 is too large to be drawn here. Figure 4 shows steps of the algorithm on a smaller example.



W_{G1}	C	shortest path for this W_{G1}?	$W_{G2} = W_G - W_{G1}$	Q_C
3	1+4=5	✓	7-3=4	5
4	1+2=3	✓	7-4=3	$\boxed{4}$
4	3+4=7		7-4=3	
5	3+2=5	✓	7-5=2	5

Figure 4: Example: A) planar DAG G with vertex weights and edge costs. B) G after moving weight values to edges. C) dual graph G^* with edge costs and edge weights. D) expanded graph G' with edge costs. E) optimum cut.

Complexity: Creation of graph G' is the most intensive phase of the algorithm. For a task graph with N vertices, the expanded graph G' has $O(N.W_G)$ vertices. Finding the single-source shortest-path problem on G' has the same complexity, because the number of edges in G' is $O(N.W_G)$. The complexity of our exact algorithm is pseudo-linear, due to its dependency on workload *values*. In other words, its runtime would change for the same task graph, if the static schedule, tasks internal computation or target architecture is modified. The amount of required memory is also proportional to the number of vertices in the expanded graph G', and thus, its memory requirement is also $O(NW_G)$.

4. APPROXIMATE TASK ASSIGNMENT

In this section, we present an approximation method for task assignment with strictly-polynomial complexity. The approximation algorithm takes as input a *tolerable error bound*, ϵ, and guarantees that solution quality is not degraded beyond the bound. In other words, throughput of the *near-optimal* solution is not more than a factor of $1+\epsilon$ worse than the optimal throughput.

We reduce the complexity by simplifying graph G' in the exact algorithm. The number of vertices in the expanded graph G' is reduced from $O(N.W_G)$ to $O(N \log W_G)$ by judiciously trimming all the W_G possible values of w to only $\log W_G$ distinct numbers. For this purpose we need an approximation function $y = f(w)$ to return a representative value y for a range of w values.

DEFINITION 4.1. *The approximation function $y = f(w)$ is defined as $f(0) = 0$, $f(w) = (1+\delta)^{\lfloor \log_{1+\delta}^{w} \rfloor}$, $\delta > 0$*

We apply this function whenever a new edge is to be added to graph G'. Thus, for a path P from ν_s to ν_t in G^*, the approximation function is applied K times, where K is the number of edges in path P. Figure 5 shows an example in which, path P is marked with dashed lines. All paths start from ν_{s_0}. In path P, weight of the edge from ν_s to ν_m is equal to 1, and based on Figure 5.b, $f(0+1) = 1$, therefore the first edge of this path is from ν_{s_0} to ν_{m_1}. Now, the next edge starts from ν_{m_1}. $1+4=5$ and $f(5) = 4$, therefore, the second edge is from ν_{m_1} to ν_{t_4} (Figure 5.c). As a result W_P of this path is equal to 4 which is an approximation of its original value $W_P = 5$ (Figure 5.d). We denote the approximated W_P with Ψ_P.

Therefore, the number of vertices in graph G' is $O(N \log_{1+\delta}^{W_G})$ because the above approximation function will result in one of the following possible distinct numbers for y: 0, 1, $1+\delta$, $(1+\delta)^2$, ..., $(1+\delta)^{\lfloor \log_{1+\delta}^{W_G} \rfloor}$.

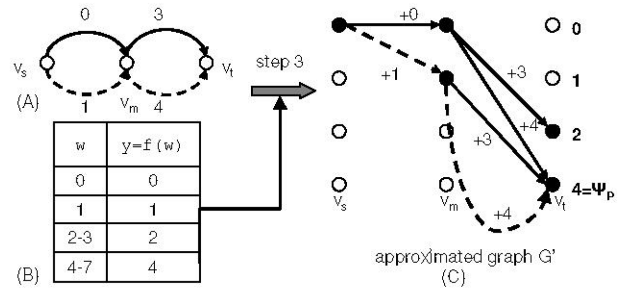


Figure 5: Example: A) dual graph G^* from Figure 4, B) an illustrative approximation function with $1+\delta = 2$. C) the resulted graph G'.

LEMMA 4.1. *For approximation function $y = f(w)$ described in Definition 4.1, $\frac{w}{1+\delta} < y \leq w$*

PROOF: $(1+\delta)^{\lfloor \log_{1+\delta}^{w} \rfloor} \leq (1+\delta)^{\log_{1+\delta}^{w}} < (1+\delta)^{\lfloor \log_{1+\delta}^{w} \rfloor + 1}$, hence based on definition of approximation function we have $y \leq w < (1+\delta)y$, and thus $\frac{w}{1+\delta} < y \leq w$. ■

THEOREM 4.1. *For every path P from ν_s to ν_t in the expanded graph G', the approximated weight Ψ_P is within the following range from its original value W_P, where K is the number of edges in path P. $\frac{W_P}{(1+\delta)^K} < \Psi_P \leq W_P$*

Intuitively, since we use the approximation function K number of times for a path P, we add the above error $(1+\delta)$ not once but K times.

COROLLARY 4.1. *If we set $\delta = \frac{\epsilon}{2F}$ where $0 < \epsilon < 1$ and F is the number of faces in task graph G:*

$$\frac{W_P}{1+\epsilon} < \Psi_P \leq W_P$$

This means that for every path P from ν_s to ν_t in the expanded graph G', the approximated weight Ψ_P is within the above range from its original exact value W_P.

THEOREM 4.2. *Let $\Omega_P = max(\Psi_P, C_P, W_G - \Psi_P)$ denote approximated value of our original cost function $Q_P = max(W_P, C_P, W_G - W_P)$. We have*

$$Q_P < \Omega_P \leq Q_P(1+\epsilon)$$

For brevity we omit details of the proof. In short, we first ignore the C_P part of function Q_P and consider two cases, $Q_P = W_P$ iff $W_P > W_G - W_P$, and $Q_P = W_G - W_P$ iff $W_P < W_G - W_P$. In both cases we apply Corollary 4.1 and eventually prove that $Q_P < \Omega_P \leq Q_P(1+\epsilon)$ is true in both cases. Then we add C_C into the equations and will see that it does not change the result.

The above theorem states that the error in calculating the cost function is bounded within a factor of $1+\epsilon$. Therefore, the near-optimum solution found in the approximated graph G' is not more than a factor away from the optimum solution which we can find in the original graph G'.

5. EVALUATION AND DISCUSSION

Setup and Methodology: Our evaluation is based on *measurements* of application throughput on actual hardware. We use Digilent XUP Virtex-II PRO FPGA board to prototype single and dual core architectures. Xilinx MicroBlaze soft processors are used as processor cores. MicroBlaze is a MIPS-based 32-bit, in-order, single issue soft processor whose architectural parameters can be configured. In our experiments, processors have a FPU, an integer divider/multiplier, and sufficient on-chip memory to contain both data and instructions. Inter-processor FIFO communication channel is implemented using Xilinx 32-bit Fast Simplex Links (FSL) with buffer size of 1024 words. Processors and FSL both run at 100MHz.

We utilize MIT StreamIt [11] compiler to evaluate our algorithms. StreamIt is a language and open-source compilation framework for static-rate streaming applications. Its compiler takes as input an application specified in synchronous dataflow (SDF) semantics with StreamIt syntax, and after static scheduling and partitioning of the graph, generates parallel `C` codes for the target architecture. Parallel codes should be compiled for the target uni-processor to generate executable binary.

One step of the aforementioned compilation flow is to partition the application graph to assign tasks to processors. We implement our algorithm (TAP) within StreamIt to replace its built-in task assignment algorithm, while utilizing its static scheduling and code generation capabilities.

The generated parallel C codes are compiled and loaded into MicroBlaze processors. Subsequently, applications throughputs are measured during execution.

Task Graph Composition and Planarization: Semantics of StreamIt are closely related to that of synchronous dataflow (SDF) graphs: Task-level parallelism is explicitly specified in semantics, and tasks internal computations are specified in a sequential C-like language. In addition, tasks are composed using a few basic guidelines. Specifically, tasks are referred to as filters, where a filter node has one input edge and one output edge. A number of filter nodes can be composed to form larger filters, according to one of the following composition rules: 1) Pipeline, 2) SplitJoin and 3) FeedbackLoop.

Pipeline implements a chain of filters in which, a node gets its input from previous filter and passes its output to next node. SplitJoin is used to specify independent data-parallel or task-parallel streams that diverge from a common splitter and merge into a common joiner. SplitJoin is similar to scatter-gather operator in parallel computing. FeedbackLoop also has a splitter and a joiner, but the joiner appears first in dataflow to join input with the output of the feedback path. Application graphs that are hierarchically composed using Pipelines, SplitJoins, and FeedbackLoop composition rules will belong to class of series-parallel graphs, and hence are planar by construction [4].

Although our algorithm requires application task graphs to be planar, this is not a real impediment to its practicality. StreamIt task graphs are planar by construction, which implies that many existing streaming applications are, or can be, modeled with planar task graphs. In addition, our proposed method is applicable to other programming language in which, specified applications can be non-planar. We present a simple transformation to temporarily planarize the graph, and revert it back after task assignment.

Imagine a particular embedding of a non-planar task graph on the 2-D plane. There are at least two edges crossing. Let us insert a dummy node, with zero workload, at the crossing point to temporarily eliminate this crossing. This procedure can be repeated for all crossings to get a temporarily planarized graph, with some dummy nodes. The planarized graph is partitioned using our algorithm, and subsequently, dummy nodes are removed from the graph. Proof and examples are eliminated due to page limitation.

Workload and Communication Estimation: We profiled MicroBlaze processor to estimate its CPI (cycle per instruction) distribution. Subsequently, tasks internal computations are analyzed at high-level, and a rough mapping between high-level language constructs and processor instructions is determined. The mapping is guided and verified by comparison with generated assembly for the processor. For SDF-compliant streaming applications, control-flow characteristics are minimal. As a result, we employed first order estimation techniques such as average if-then-else path latencies, and expected number of loop iterations, whenever needed. The analysis derived w'_v, which represents clock cycles needed for every firing of node v.

Computing inter-task communication cost is simpler due to our application model. For applications modeled in SDF, each node appears a specific number of times in the steady state schedule. Assume node v is fired $n(v)$ times in an execution period. Note that $n(v)$ is calculated statically for SDF applications [7, 8]. The number of data samples produced and consumed per firing of each node is also specified at compile time. Let $p(uv)$ denote the number of data samples sent from u to v, every time u is fired. It follows that:

$$w(v) = n(v) \times w'(v), \text{ and } c_{uv} = \frac{n(v) \times p(uv)}{B}$$

Where $w(v)$ is estimated workload of node v, and c_{uv} is estimated communication latency from task u to v, in case they are assigned to different processors. B denotes the bandwidth of the inter-processor channel.

Testbenches: Figure 6 shows the benchmarks used in our experiments. They represent commonly used streaming applications that are typically utilized in portable, multimedia and signal processing embedded systems. The applications are selected from the StreamIt benchmark set, having in mind the data and instruction memory constraints of our FPGA board. The last three columns show the number of vertices, edges and faces of the task graph (equal to vertices in dual graph).

Application	Description	Task Graph Structure		
		V	E	F
BSORT	Bitonic Sort	756	1012	259
MATMUL	Blocked Matrix Multiply	23	23	3
FFT	Fast Fourier Transform	152	207	58
TDE	Freq. Domain Convolution	46	52	9
FILTER	Discrete Filter	53	59	9

Figure 6: Benchmark applications.

Results and Discussion: Our first experiment results, depicted in Figure 7, compares applications throughput using StreamIt and our exact task assignment (TAP) algorithms. Throughput is measured as the number of outputs (data samples) per second produced by single- and dual-core hardware. Columns 3 and 4 represent the normalized throughput values with respect to a uni-processors. All produced data samples are 4 bytes in our applications. In all cases, TAP matches or outperforms StreamIt. The improvements are as high as 12.6%, and 6.4% on average.

Application	StreamIt	TAP	Strea. vs. uni-proc (norm.)	TAP vs. uni-proc (norm.)	TAP vs. Strea. (%)
BSORT	296.3	319.2	1.58	1.70	7.7
MATMUL	186.3	208.0	1.38	1.55	11.6
FFT	417.7	470.4	1.58	1.77	12.6
TDE	933.8	933.8	1.61	1.61	0.0
FILTER	34.6	34.6	1.88	1.88	0.0
Average			1.61	1.70	6.4

Figure 7: Comparing measured throughput of StreamIt and TAP task assignment.

Application	Runtime (second)	Memory Consumption (MB)	Throughput (outputs per sec.)
BSORT	31.8	2543	319,200
MATMUL	57.5	321	208,000
FFT	46.7	2553	470,400
TDE	76.6	844	933,800
FILTER	121.5	1366	34,640

Figure 8: Runtime and memory requirement for optimal task assignment.

Recall that TAP runs in pseudo-polynomial time in problem size. Its runtime and memory requirement depend on the value of estimated workload. TAP's runtime and memory demand is quite reasonable for small benchmarks. To experiment the effectiveness of our approximate task assignment algorithm, we intentionally inflate workload estimation by two orders of magnitude. This pronounces the inability of exact algorithm to scale arbitrarily. Note that inflating workload estimation preserves their relative intensity, and ideally, should not affect task assignment quality.

Figure 8 shows the time and memory required to run the exact task assignment algorithm, along with the optimal throughput. Interestingly, the runtime for BSORT, which has the most number of tasks, is the smallest. Also, it takes over two minutes for FILTER, a small application with only 53 tasks. This is because TAP runtime is a strong function of total workload, which does not necessarily correlate with number of tasks. Total workload also depends on intra-task computation and static schedule. Moreover, TAP allocates up to 2.5 GB of memory, which impedes its utilization in many systems.

Epsilon	Runtime(%)	Memory(%)	Throughput(%)
		BSORT	
0.1	18.2	1.8	99.2
0.5	14.5	0.9	98.7
0.9	13.8	0.7	98.7
		MATMUL	
0.1	41.4	20.0	100
0.5	40.9	20.0	100
0.9	40.2	19.9	95.3
		FFT	
0.1	34.5	1.8	100
0.5	33.8	1.7	98.1
0.9	33.4	1.7	93.2
		TDE	
0.1	36.9	9.2	100
0.5	36.9	9.1	93.3
0.9	36.8	9.1	92.6
		FILTER	
0.1	41.4	9.1	95.2
0.5	41.1	9.1	91.9
0.9	37.9	9.1	93.9

Figure 9: Normalized runtime, memory and throughput for selected approximation bounds.

Subsequently, we apply our approximate task assignment algorithm to trade throughput for algorithm time and memory requirement. Figure 9 shows the same parameters as Figure 8 for the near-optimal solution offered by the approximate algorithm. Throughput, runtime and memory values are normalized with respect to their values in Figure 8. For example, in BSORT with $\epsilon = 0.1$, the approximate algorithm finds a near-optimal assignment with 99.2% throughput of the exact algorithm, while it consumes only 1.8% memory and 18.2% time. Due to page limitation, we report the results for only three values of ϵ.

The approximation bound (ϵ) serves as a *knob* for designers to adaptively favor throughput over time and memory consumption. In our experiments, application graphs are small and therefore, loosening the bound does not have a large impact on solution quality or optimization cost. In other words, the trimmed graphs at $\epsilon = 0.1$ and $\epsilon = 0.9$ look very similar. There are two important points to notice here: Firstly, as expected, the results are within the proved bound in all cases. Secondly, for a given application, throughput at a larger ϵ does not have to be worse than throughput at a smaller ϵ. Approximation bound only guarantees a lower bound on quality loss, but it does not provide provably monotone quality degradation.

Figure 10 visualizes the geometric mean of throughput-memory and throughput-runtime tradeoff points, over all ϵ values and applications. Note that due to space limitation, Figure 9 only reports data for three ϵ values. On average, finding the near-optimal solution requires 30.4% to 33.1% time, 4.6% to 5.6% memory, and results in 94.7% to 98.9% throughput, compared to the optimal solution.

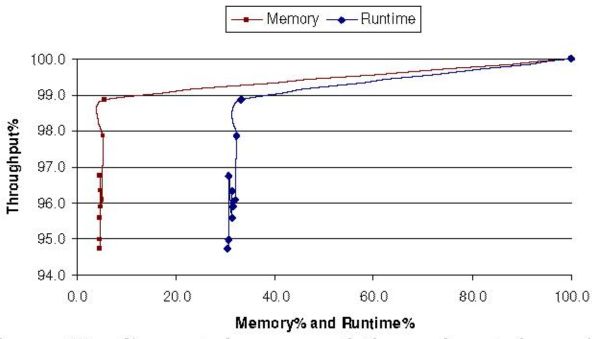


Figure 10: Geometric mean of throughput degradation vs. runtime improvement.

6. CONCLUSIONS

We presented a methodology for synthesizing streaming applications on embedded dual-core architectures. We developed a task assignment algorithm and proved its optimality in maximizing the throughput. Also, we devised an approximation method to reduce the runtime and memory consumption from pseudo-linear to linear. We proved that the quality of the near-optimal solution remains within a factor of $1+\epsilon$ from the optimal solution, for arbitrary ϵ values. We measured the application throughput on operating hardware. Measurements validated our mathematical contributions. On average, the approximate method runs about 3 times faster, requires only about 5% memory, and results in throughput loss of about 1% to 5%.

7. REFERENCES

[1] K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] S. S. Bhattacharyya et al. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, 1999.

[3] J. Cong et al. Synthesis of an application-specific soft multiprocessor system. In *FPGA*, 2007.

[4] M. I. Gordon et al. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

[5] M. I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[6] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA*, 2005.

[7] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[8] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[9] J. Parkhurst et al. From single core to multi-core: preparing for a new exponential. In *ICCAD*, 2006.

[10] M. B. Taylor et al. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA*, 2004.

[11] W. Thies et al. Streamit: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction*, 2002.