Versatile Task Assignment for Heterogeneous Soft Dual-Processor Platforms

Matin Hashemi, *Student Member, IEEE,* and Soheil Ghiasi, *Member, IEEE*

***Abstract*—Heterogeneous soft multiprocessor systems are likely to find a larger share in the application-specific computing market due to increasing cost and defect rates in foreseeable manufacturing technologies. We study the problem of mapping streaming applications onto heterogeneous soft dual-processor systems, in which processors' limited memory resources and application throughput form the outstanding constraints and objective, respectively. A key step in the compilation process is task assignment, where tasks are assigned to the processors. We develop a provably-effective algorithm for task assignment. Our algorithm is versatile, in that its formal properties hold for, and hence it is applicable to, a variety of platforms. Measurement of generated code size, and throughput of emulated systems validate the effectiveness of our approach. We advance the state-of-the-art by considerably outperforming two recent competitors in terms of both versatility and application throughput.**

***Index Terms*—FPGA synthesis, multiprocessor SoC, optimization, system-on-chip.**

I. Introduction

BOTH EXISTING and viable alternative manufacturing technologies are expected to have increasingly higher costs and defect rates. As a result, programmable substrates are likely to replace dedicated application-specific solutions in a broader range of applications [1]. In such a scenario, application-specific solutions would be implemented as *soft* architectures, in that, they are realized out of pre-designed programmable building blocks. Outstanding examples in today's technology are soft processors in commodity field-programmable gate arrays (FPGAs).

The limited area of a programmable platform should be judiciously divided among storage, processing, or communication resources to holistically optimize application performance [2]–[4]. For instance, a processor that runs integer code can do without a floating point unit to free up its area for other resources in the system.

We study the problem of mapping streaming applications onto such heterogeneous soft multiprocessor systems. In mapping the applications, we aim to optimize throughput, which is an important quality metric in the streaming application space. Simultaneously, we ensure that generated code for a processor meets processor's instruction and data memory size constraints, if needed. Given the high-degree of heterogeneity in the system, different architecture customizations are likely to impact application throughput differently. Optimal mapping of application tasks onto processors depends on the specifics of the architectures. Ideally, one would want to have a *versatile* software synthesis solution that can be parameterized to target different configurations.

In this paper, we contribute to such a software synthesis solution by presenting a formally-versatile task assignment algorithm for soft dual-processors. Leveraging the graph theoretical properties of applications task graphs, we develop a provably-effective task assignment algorithm that both maximizes throughput, and guarantees meeting instruction and data size constraints according to high-level estimates. Our technique is extensible in that it can handle similar implementation-driven objectives and constraints. Experimental evaluation of implemented applications validates the efficacy of our approach.

II. Related Work

A number of recent efforts focus on multiprocessor software synthesis for synchronous dataflow (SDF) streaming applications. Thies *et al.* [5] present StreamIt, a modeling language and compiler for streaming applications. Gordon *et al.* [6] and Thies *et al.* [7] describe a task assignment method to partially explore task parallelism for homogeneous platforms. Gordon *et al.* [8] extend their work by a heuristic algorithm for acyclic StreamIt task graphs to exploit task, data, and pipeline parallelism. As part of the Ptolemy project, Pino *et al.* [9] propose a combined task assignment and scheduling method for acyclic SDF graphs on homogeneous platforms.

Stuijk *et al.* [10] propose a task assignment method for heterogeneous systems. Tasks are first sorted based on their impact on throughput, then a greedy method assigns one task at a time to the processor with the least workload. Cong *et al.* [11] present an algorithm for assignment of acyclic task graphs onto application-specific soft multiprocessors. It starts by labeling the tasks, followed by clustering them into processors, and finally, tries to reduce the number of processors by packing more tasks onto under-utilized processors.

In [12], [13], we presented a task assignment algorithm that unlike other methods delivers provably-optimal solutions. We exploit planarity of target task graphs to achieve a reasonably low complexity. Some SDFs, such as those specified in StreamIt, are inherently planar. In addition, we developed a

Manuscript received February 17, 2009; revised June 27, 2009 and September 3, 2009. Current version published February 24, 2010. This paper was recommended by Associate Editor V. Narayanan.

The authors are with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616 USA (e-mail: hashemi@ucdavis.edu; ghiasi@ucdavis.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2010.2041856

planarization transformation to handle non-planar task graphs. This paper advances our work by exact handling of heterogeneous processors and memory size constraints; versatility in targeting various platform customizations; and corresponding formal analysis, e.g., approximation method.

III. Preliminaries

A. Application Model

We require the application to be modeled as a set of explicitly-parallel communicating tasks. Task graph models address the scalability and productivity issues of parallel threads [14]. In this model, an application is represented with a directed graph whose vertices are tasks and the edges are inter-task data dependences. Tasks are atomic, i.e., their corresponding computation is specified in sequential semantics, and hence, intra-task parallelism is not exploited.

There are many variations of task graphs [15], [16]. For example, in Kahn process networks, tasks communicate via unidirectional first in, first out (FIFO) links, and receiving tasks block on empty input links [17]. SDF can also be viewed as a task graph in which vertices have to fire a certain number of times in the periodic schedule [18].

We target task graphs that contain no cycles. If a given task graph contains a cycle, we collapse the vertices in the cycle into a single task to represent the application as a directed acyclic graph (DAG). Such task graphs form an important subset, because many important streaming kernels can be represented in this fashion (Section X).

B. Target Platform

Our software synthesis scheme views processors to have dedicated memory spaces, and communicate via sending and receiving messages [Fig. 1(b)]. Note that this is merely a *virtual* view of hardware, and not necessarily the physical implementation. Systems with shared on-chip memory can be programmed to implement the virtual inter-processor message communication in the shared memory space, for example, with a virtual FIFO as an array in memory.

The virtual network can be realized on the chip in a number of ways, e.g., point-to-point links or on-chip network. The implemented communication mechanism has to deliver messages in the order they are sent. Note that message delivery in deterministic order, e.g., in-order delivery, is required for correct realization of statically scheduled applications.

C. Software Synthesis

We overview our software synthesis scheme with a simple example. Fig. 1(a) illustrates a toy streaming application that calculates `cos` of the angle between two input vectors, each with three components x, y, and z. It continuously reads vectors from the input stream, calculates $t = \cos(\alpha)$, and writes the results into the output stream.

Our goal is to automatically synthesize the application code for execution on our soft dual-processor target shown in Fig. 1(b). As an intentional result of our application and architecture models, synthesized software modules need to directly send and receive messages to synchronize.

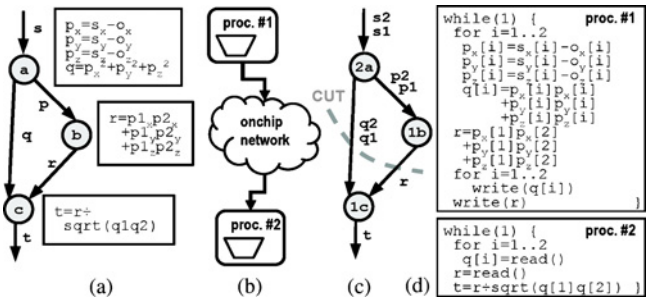


Fig. 1. Example application: $t = \cos(\alpha) = \frac{(\vec{s1}-\vec{o}).(\vec{s2}-\vec{o})}{|\vec{s1}-\vec{o}||\vec{s2}-\vec{o}|}$. (a) Task graph. (b) Soft dual-processor platform. (c) Tasks are scheduled at $2a1b1c$, i.e., $2a$ means task a runs twice. The cut in the graph implies an assignment of tasks to processors. (d) Synthesized software for depicted task assignment.

We statically determine tasks' schedule based on the production and consumption rates. Task `b` reads two consecutive vectors coming from task `a`, i.e., `p1` and `p2`. Since, `a` produces one token while `b` needs two, `a` has to run twice as many times as `b`. Therefore, the periodic schedule is `2a1b1c`.

Next, each task is assigned to one of the processors. In our example, tasks `a` and `b` are assigned to processor 1 [Fig. 1(c)]. Given a set of tasks assigned to processor p, the corresponding software module is synthesized by combining computations of tasks according to the given schedule, complemented by calling architecture-dependent `read` and `write` primitives [Fig. 1(d)]. We use the original task graph schedule to determine the local ordering of tasks assigned to each processor. For example, tasks `b` has to come after `a` because the original schedule was `2a1b1c`.

IV. Problem Statement

Given the application and architecture models, and tasks' static schedule, we aim to assign tasks to processors such that the synthesized software modules: 1) fit in the possibly-limited memory resources of soft processors; and 2) implement the application with maximum steady state throughput.

In addition to computation workload of processors, inter-processor communications impact application throughput as well. For example, in a system with a slow network, one may want to assign two tasks with large inter-task communications to the same processor in order to avoid overloading the network. In subsequent sections, we will discuss the specific relation between tasks' workload and inter-task communication requirement, and application throughput.

V. Terms and Definitions

A. Vertex and Edge Attributes

As summarized in Fig. 2, we annotate the task graph with a number of *attributes*. Later in Section IX, we discuss extensions to handle other attributes, if necessary.

1) *Computation:* Attribute $w_i(v)$ refers to the number of clock cycles it takes for processor i to execute task v. In Fig. 1(a), $w_i(c)$ is the latency of one multiplication, one square root, and one division on processor i.

2) *Communication:* Attribute $n(e)$ refers to the number of messages communicated over edge e between two adjacent tasks. Unlike w and m attributes, n is not implementation

dependent. For a given task graph, it is constant and known at compile time. In Fig. 1, $n(ab) = 2 \times |p|$ because there are two consecutive vectors `p1` and `p2` on this edge.

3) *Memory:* Attribute $m_i(v)$ refers to the number of bytes required by processor i for instructions of task v. In Fig. 1, $m_i(c)$ is the memory footprint of one multiplication, one square root, and one division on processor i. Similarly, $m_i(e)$ refers to the number of bytes allocated by the array which stores messages of edge e. In Fig. 1, $m_1(ab)$ is equal to the amount of memory allocated on processor 1 by messages on edge ab, i.e., $m_1(ab) = n(ab) \times \text{sizeof}(float)$.

B. Partition and Partition Attributes

Task assignment for dual-processor platforms can be viewed as bi-partitioning of application task graph, in which tasks in the same partition are assigned to the same processor. A partitioning of graph $G(V, E)$ is constructed by removal (cutting) of a subset of edges to create two connected subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$. We use the term cut C to refer to such cut edges. The aforementioned attributes are naturally extended to partitions in the following manner (Fig. 3).

1) *Computation:* W_i refers to the total computation workload of tasks in a partition when assigned to processor i. In Fig. 1, $V_1 = \{a, b\}$ and $V_2 = \{c\}$, and thus, $W_1(V_1) = w_1(a) + w_1(b)$ and $W_2(V_2) = w_2(c)$. For a subset of tasks, V_i, assigned to processor i, $W_i(V_i) = \sum_{v \in V_i} w_i(v)$.

2) *Communication:* If an edge $e(u, v)$ is cut, tasks u and v are assigned to different processors, and thus the corresponding communication between u and v has to pass through the virtual on-chip network. The term $N(C)$ to refers to the number of messages transfered over all the cut edges. In Fig. 1, $C = \{ac, bc\}$ and thus $N(C) = n(ac) + n(bc)$.

3) *Memory:* Cut edges demand allocation of data memory on both processors. The term $M_i(C)$ refers to the amount of data memory allocated for the cut edges on processor i. In Fig. 1, $M_1(C) = m_1(ac) + m_1(bc) = n(ac) \times \text{sizeof}(float) + n(bc) \times \text{sizeof}(float)$. Formally, $M_i(C) = \sum_{e \in C} m_i(e)$.

The rest of the edges allocate memory in only one processor. The term $M_i(E_i)$ refers to the total amount of memory required by processor i for implementing intra-processor communication, in which E_i represents the subset of edges that are allocated to processor i. In Fig. 1, $E_1 = \{s, p\}$, and thus, $M_1(E_1) = (2|s| + 2|p|) \times \text{sizeof}(float)$. Similarly, the term $M_i(V_i)$ denotes the amount of instruction memory required by processor i for task set V_i. Formally, $M_i(E_i) = \sum_{e \in E_i} m_i(e)$ and $M_i(V_i) = \sum_{v \in V_i} m_i(v)$.

Note that since task graph G is acyclic, we aim to assign tasks to processors such that one processor would always send and the other would always receive message. This unidirectional flow of data intuitively suits pipelined throughput-driven execution of streaming applications, while guaranteeing that original static schedule of tasks remains feasible after partitioning [4]. In the graph domain, this translates to *convex* cuts, which refer to cuts that connect vertices in G_1 to vertices in G_2 (and not the other way around). Convex cuts, which form the focus of our work, characterize a subset of possible task assignments. If all task assignment possibilities were to be considered, task scheduling had to be combined with task assignment to allow evaluation of candidate solutions.

Attribute	Notation	Unit
computation workload	$w_i(v)$	clock cycle
number of messages	$n(e)$	-
required memory	$m_i(v), m_i(e)$	byte

Fig. 2. Attributes of vertices and edges of a task graph, e.g., $w_2(v)$ is the number of cycles it takes for proc. 2 to execute task v (see Section V-A).

$W_i(V) = \sum_{v \in V} w_i(v)$	computation workload assigned to processor i
$N(C) = \sum_{e \in C} n(e)$	# of message transferred between processor 1 and 2
$M_i(V_i) = \sum_{v \in V_i} m_i(v)$ $M_i(E_i) = \sum_{e \in E_i} m_i(e)$ $M_i(C) = \sum_{e \in C} m_i(c)$	memory required by processor i for: - tasks - inter-task intra-processor communications - inter-processor communications

Fig. 3. Partition attributes are natural extension of vertex and edge attributes.

C. Problem Statement

Tasks in the partition G_i will be assigned to, and executed by processor i, and messages on edges $e \in C$ (cut edges) will be transfered between processors using the virtual onchip network (Fig. 1). Hence, task assignment for dual-processors can be formally defined as the following optimization problem:

$$\begin{aligned} &a) \text{ minimize:} \\ &\quad Q = F\big(W_1(V_1), W_2(V_2), N(C)\big) \\ &b) \text{ constraint:} \\ &M_1(V_1) + M_1(E_1) + M_1(C) \le \text{memory of processor 1} \\ &M_2(V_2) + M_2(E_2) + M_2(C) \le \text{memory of processor 2.} \end{aligned} \tag{1}$$

That is, the goal is to find the cut C which a) maximizes throughput; and b) meets the processors' memory size constraints. The cost function Q models application execution period, which is the inverse of its throughput. The execution period depends on processor workloads and inter-processor communication. The specific relation, however, depends on the underlying hardware. Our goal is to devise a versatile method to handle a wide variety of cost functions.

For example, in a system with negligible interprocessor communications, it may be accurate enough to estimate the execution period as $Q = F(W_1, W_2) = \max\{W_1, W_2\}$, because the pipeline throughput would be limited by the slowest of the two processors. This simplified cost function promotes balancing processors' workload, which has been the focus of most classical task assignment schemes. As another example, assume a sufficiently-buffered FIFO link between the two processors implements the virtual network. A reasonable estimation function would be $Q = \max\{W_1 + \alpha_1 N, \alpha_2 N + W_2\}$, where α_i is the extra cycles in workload of processor i to push (pop) one unit of data to (from) the FIFO.

In practice, out of two solutions with identical workload distributions, the one with smaller inter-processor communication is always preferred. Therefore, without loss of generality we assume that the cost function Q is non-descending in N.

VI. Attribute Properties and Transformations

In this section, we exploit structural properties of task graphs to develop a transformation on the attributes, which assists us in quick evaluation of a cut. Let us assume an imaginary attribute θ_1 is assigned to both vertices and edges of graph $G(V, E)$. Our objective is to transform θ_1 to a new set of attributes, θ'_1, assigned only to edges of G, such that their summation on any cut would give the summation of the original θ_1 attributes on vertices and edges in partition 1.

Let e_i and e_o denote the set of incoming and outgoing edges of a vertex, respectively. Let e^*_o be a randomly-selected outgoing edge. The transformed attributes θ'_1 for outgoing edges of a vertex are recursively defined as follows:

$$\forall v \in V \ : \ \theta'_1(e_o) = \begin{cases} \sum_{e_i} \theta'_1(e_i) \ + \theta_1(v) + \theta_1(e_o) & \text{if } e^*_o \\ \theta_1(e_o) & \text{otherwise.} \end{cases}$$

Fig. 4 shows an example. Intuitively speaking, attribute transformation works similar to gravity. The task graph can be viewed as a physical structure in which every element has a weight, i.e., $\theta_1(v)$ and $\theta_1(e)$ can be viewed as the weight of vertex v and edge e, respectively. Hence, the random selection of one outgoing edge is analogous to disconnecting the corresponding joint in the structure. That is, every vertex stays connected to exactly one of its outgoing edges in the physical domain. The amount of weight held by edge e determines the value of $\theta'_1(e)$. In Fig. 4, edge ab holds $\theta'_1(ab) = \theta_1(a) + \theta_1(ab)$ of weight. Vertex b in the figure is disconnected from edge bc, and therefore, total weight of vertices a and b and edges ab and bd is held by edge bd. Note that this joint disconnection is only an intuitive analogy, i.e., none of the edges are actually removed or disconnected.

As expected from the gravity analogy, the transformation satisfies our objective property. That is, the sum of θ'_1 attributes on cut C is equal to the sum of θ_1 attributes on the edges and vertices of the top partition of graph, $G_1(V_1, E_1)$. Intuitively, the cut edges have to hold the entire weight above them. In Fig. 4, $\theta'_1(bd)+\theta'_1(ce)$ is equal to $\{\theta_1(a)+\theta_1(b)+\theta_1(c)\}+\{\theta_1(ab)+\theta_1(bc)\} + \{\theta_1(bd) + \theta_1(ce)\}$.

Theorem 1: For a convex cut on a directed acyclic graph[1]

$$\underbrace{\sum_{v\in V_1} \theta_1(v)}_{\substack{\text{weight of} \\ G_1 \text{ vertices}}} + \underbrace{\sum_{e\in E_1} \theta_1(e)}_{\substack{\text{weight of} \\ G_1 \text{ edges}}} + \underbrace{\sum_{e\in C} \theta_1(e)}_{\substack{\text{weight of} \\ \text{cut edges}}} = \underbrace{\sum_{e\in C} \theta'_1(e)}_{\substack{\text{total weight} \\ \text{held by cut } C}} .$$

The theorem should be intuitive from the gravity analogy, in which the weight of every vertex or edge in G_1 is held by one and only one of the cut edges. For example, in Fig. 4, weight of vertex a [i.e., $\theta_1(a)$] is held by the cut edge bd, and not by ce. Therefore, across all the cut edges, weight of vertices and edges in G_1 are considered exactly once.

We also introduce another transformation similar to above, with the minor difference that the gravity is replaced with a force away from (as opposed to toward) the bottom of the graph. In other words, the graph can be temporarily held upside down. Here, we convert θ_2 attributes to a new set of θ'_2 attributes assigned to the edges. Fig. 5 shows an example.

[1]All proofs are omitted from the paper due to space limitation. They are available in our group webpage at `leps.ece.ucdavis.edu` under the Publications menu.

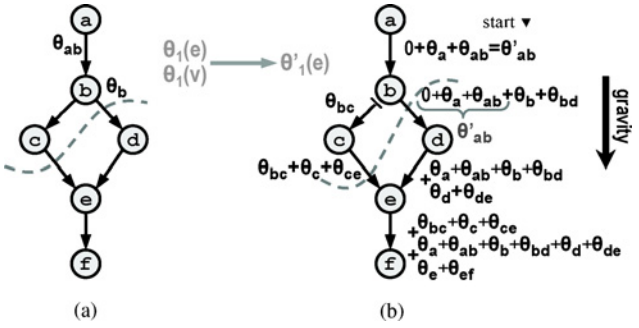


Fig. 4. (a) Sample task graph with attributes θ_1 for vertices and edges. (b) Task graph after transforming θ_1 to θ'_1.

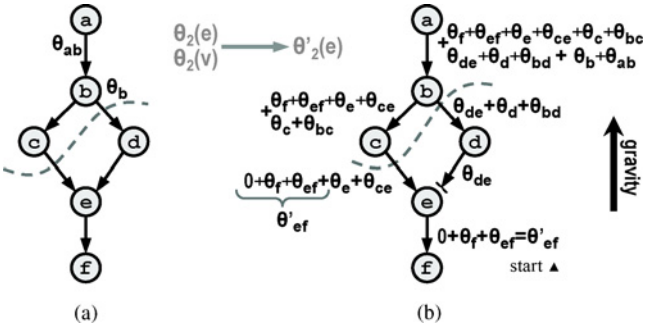


Fig. 5. (a) Sample task graph with attributes θ_2 for vertices and edges. (b) Task graph after transforming θ_2 to θ'_2.

Theorem 2: Similarly we have

$$\underbrace{\sum_{v\in V_2} \theta_2(v)}_{\substack{\text{weight of} \\ G_2 \text{ vertices}}} + \underbrace{\sum_{e\in E_2} \theta_2(e)}_{\substack{\text{weight of} \\ G_2 \text{ edges}}} + \underbrace{\sum_{e\in C} \theta_2(e)}_{\substack{\text{weight of} \\ \text{cut edges}}} = \underbrace{\sum_{e\in C} \theta'_2(e)}_{\substack{\text{total weight} \\ \text{held by cut } C}} .$$

Corollary 1: By combining Theorems 1 and 2 we have the following formula for $i = \{1, 2\}$:

$$\sum_{v\in V_i} \theta_i(v) + \sum_{e\in E_i} \theta_i(e) + \sum_{e\in C} \theta_i(e) = \sum_{e\in C} \theta'_i(e).$$

VII. Versatile Task Assignment via Graph Partitioning

We approach task assignment as a graph partitioning instance, and develop an algorithm that is provably optimal in minimizing cost function Q, or equivalently, maximizing pipeline throughput. The technique is versatile and can optimize a variety of cost functions inspired by different hardware configurations. For clarity, we present our technique using the attributes discussed in Section V. Later in Section IX, we will present extensions to our approach.

A. Applying Attribute Transformation

As summarized in Fig. 3, to obtain the value of partition attributes all vertices or edges in the partition have to be enumerated. For example, one has to enumerate all vertices in G_1 to calculate W_1. It would be more efficient to evaluate

(1) by only processing attributes of the edges in cut C. Here we apply the attribute transformation, and as a result, we will have a new set of attributes (e.g., w'_1 instead of w_1), which have the desired property. The transformation is applied to the following attributes. Fig. 6 provides a summary.

1) *Computation:* w_i attributes are transformed to a new set of attributes called w'_i. In this case, "θ" is mapped to "w," which means, $\theta_i(v) = w_i(v)$, $\theta_i(e) = 0$ and $\theta'_i(e) = w'_i(e)$. Based on Corollary 1, we have

$$\underbrace{\sum_{v\in V_i} w_i(v)}_{W_i(V_i)} = \underbrace{\sum_{e\in C} w'_i(e)}_{\stackrel{\text{def}}{=} W'_i(C)}.$$

For example, in Fig. 7(a), we need to sum up the workload of vertices a, b, and c to calculate $W_1 = w_1(a) + w_1(b) + w_1(c)$ from the original graph. After the transformation [Fig. 7(b)], we can obtain W_1 by enumerating edges bd and ce, i.e., $W'_1 = w'_1(bd) + w'_1(ce) = \{w_1(a) + w_1(b)\} + \{w_1(c)\}$. Thus, we use $W'_1(C)$ instead of $W_1(V_1)$.

2) *Communication:* Because $N(C) = \sum_{e\in C} n(e)$, it is already calculated by looking only at the cut edges. Therefore, we do not need to apply the transformation to attributes n.

3) *Memory:* Attributes m_i are transformed to a new set of attributes called m'_i. Here, $\theta_i(v) = m_i(v)$, $\theta_i(e) = m_i(e)$, and $\theta'_i(e) = m'_i(e)$. Based on Corollary 1, we have

$$\underbrace{\sum_{v\in V_i} m_i(v)}_{M_i(V_i)} + \underbrace{\sum_{e\in E_i} m_i(e)}_{M_i(E_i)} + \underbrace{\sum_{e\in C} m_i(e)}_{M_i(C)} = \underbrace{\sum_{e\in C} m'_i(e)}_{\stackrel{\text{def}}{=} M'_i(C)}.$$

For example, the memory requirement of processor 2 is originally calculated by visiting vertices f, e, and d, and edges ef, de, bd, and ce, i.e., $M_2(V_2) + M_2(E_2) + M_2(C) = \{m_2(f)+m_2(e)+m_2(d)\}+\{m_2(ef)+m_2(de)\}+\{m_2(bd)+m_2(ce)\}$ [Fig. 7(a)]. After the transformation, we can derive the same value from $m'_2(bd) + m'_2(ce)$ [Fig. 7(b)]. Thus we use $M'_i(C)$ instead of $M_i(V_i) + M_i(E_i) + M_i(C)$.

Following transforming the attributes, we have a new set of edge attributes, $n(e)$, $w'_1(e)$, $w'_2(e)$, $m'_1(e)$, and $m'_2(e)$, that are summarized in Fig. 6. They enable evaluation of both the cost function and memory constraint by enumeration of the edges in cut C. Therefore, our target task assignment problem can be cast as finding the cut in the graph, subject to the following objective and constraints:

$$\begin{aligned} &a)\ \text{minimize:} \quad Q(C) = F\big(W'_1(C), W'_2(C), N(C)\big) \\ &b)\ \text{constraints:} \quad M'_1(C) \le \text{memory of processor 1} \\ &\qquad\qquad\quad M'_2(C) \le \text{memory of processor 2.} \end{aligned} \tag{2}$$

To simplify the notation, we represent the attributes in an *attribute vector* $\vec{\beta}(e) = [w'_1(e), w'_2(e), m'_1(e), m'_2(e), n(e)]$. For example, $\beta_3(e) = m'_1(e)$ and thus $\beta_3(C) = M'_1(C) = \sum_{e\in C} m'_1(e)$ [Fig. 7(c)]. Therefore, (2) can be formulated as follows:

$$\begin{aligned} &a)\ \text{minimize:} \quad Q(C) = F\big(\beta_1(C), \beta_2(C), \beta_5(C)\big) \\ &b)\ \text{constraints:} \quad \beta_3(C) \le \beta_3^{\max} \text{ and } \beta_4(C) \le \beta_4^{\max}. \end{aligned} \tag{3}$$

$\theta_i \to \theta'_i$	Corollary 6.3	Calculation
$\theta_i(v) = w_i(v)$, $\theta_i(e) = 0$; $\theta'_i(e) = w'_i(e)$	$W_i(V_i) = W'_i(C)$	$\sum_{e\in C} w'_i(e)$
transformation not required	$N(C)$	$\sum_{e\in C} n(e)$
$\theta_i(v) = m_i(v)$, $\theta_i(e) = m_i(e)$; $\theta'_i(e) = m'_i(e)$	$M_i(V_i) + M_i(E_i) + M_i(C) = M'_i(C)$	$\sum_{e\in C} m'_i(e)$

Fig. 6. Applying attribute transformation to the application task graph.

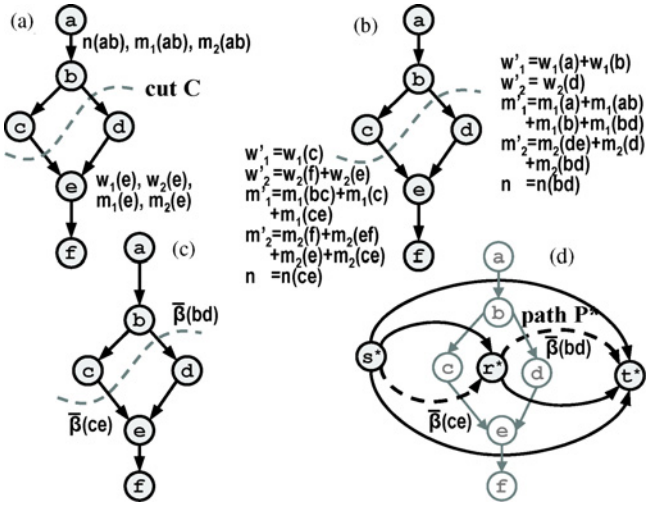


Fig. 7. (a) Task graph $G(a, f)$ and a sample cut C. (b) New attributes after the transformation. (c) Vector representation of the attributes. (d) Dual graph $G^*(s^*, t^*)$, and path P^*.

The term $\beta_3^{\max}$ (and $\beta_4^{\max}$) denotes the same value we had in (2), i.e., available memory of processor 1 (and 2).

B. Graph Expansion

The number of convex cuts C in G can grow exponentially with respect to graph complexity. In order to tractably find the optimal path, we would need to eliminate paths that are guaranteed to yield inferior solutions from consideration.

For this purpose, we first planarize the task graph G using the transformation developed in our previous work [13]. Note that although some programming languages, such as StreamIt [5], guarantee task graph planarity, our proposed method does not require a specific language.

Given a planar embedding of $G(V, E)$, we construct the well-defined dual graph $G^*(V^*, E^*)$. The β attributes of edges $e \in E$ are transferred to the corresponding edges $e^* \in E^*$ [Fig. 7(d)]. A simple path P^* from vertex s^* to t^* in graph G^* identifies a convex cut C in graph G. Therefore, the dual graph enables us to evaluate the quality of a task assignment by evaluating the quality of the corresponding path from s^* to t^* in G^*. That is, since $\vec{\beta}(C) = \sum_{e\in C} \vec{\beta}(e) = \sum_{e^*\in P^*} \vec{\beta}(e^*) = \vec{\beta}(P^*)$, we evaluate (3) from P^* instead of C

$$\begin{aligned} &a)\ \text{minimize:} \quad Q(P^*) = F\big(\beta_1(P^*), \beta_2(P^*), \beta_5(P^*)\big) \\ &b)\ \text{constraints:} \quad \beta_3(P^*) \le \beta_3^{\max} \text{ and } \beta_4(P^*) \le \beta_4^{\max}. \end{aligned} \tag{4}$$

Next, we expand the dual graph G^* to a graph $G^\dagger$, constructed in four dimensional (4-D) space, to better visualize

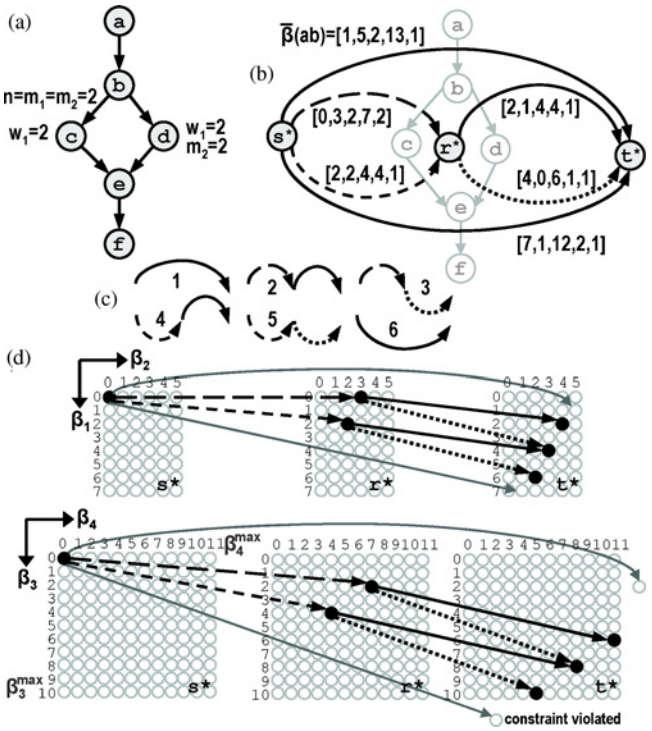


Fig. 8. (a) In task graph $G(a, f)$, all the attributes w_1, w_2, n, m_1, and m_2 are equal to 1, except for those shown in the figure. (b) Dual graph $G^*(s^*, t^*)$ and the $\vec{\beta}$ attributes. (c) Six possible paths to choose from. (d) Construction of $G^\dagger$ in 4-D space.

the situation (Fig. 8). Each dimension of $G^\dagger$ represents one element of the attribute vector, e.g., workload of the top partition. For every vertex v^* in G^*, we have vertices $v^*[i_1, i_2, i_3, i_4]$ in graph $G^\dagger$ in 4-D space. For example, in the figure, vertex r^* is expanded into two vertices $r^*[0, 3, 2, 7]$ and $r^*[2, 2, 4, 4]$ [4-D indices of the two vertices are in the middle column of Fig. 8(d)]. Graph $G^\dagger$ is constructed such that the information stored in $\vec{\beta}(e^*)$ attributes in graph G^* is represented in the structure of graph $G^\dagger$. Formally, vertex $u^*[\vec{i}]$ (abbreviation for $u^*[i_1, i_2, i_3, i_4]$) is connected to vertex $v^*[\vec{j}]$ in $G^\dagger$ when

$$\begin{cases} \text{there exists an edge } e^*(u^*, v^*) \text{ in } G^* \\ \text{and } \vec{i} + \vec{\beta}(e^*) = \vec{j} \\ \qquad\qquad \searrow \text{the first four elements.} \end{cases}$$

In other words, for a given edge e^* connecting u^* to v^* in graph G^* and a given vector $\vec{i}$, there is a unique edge in $G^\dagger$ which connects vertex $u^*[\vec{i}]$ to $v^*[i + \vec{\beta}(e^*)]$. As a result, for a given path P^* from s^* to t^* in G^*, and a given start vector $\vec{o}$, there is a unique path in $G^\dagger$ starting from $s^*[\vec{o}]$. That is, $\beta_5(u^*[\vec{i}], v^*[\vec{j}]) = \beta_5(u^*, v^*)$. Moreover, if we choose $\vec{o} = [0, 0, 0, 0]$, the path in $G^\dagger$ always ends at $t^*[\vec{\beta}(P^*)]$, because each edge from $u^*[\vec{i}]$ to $v^*[\vec{j}]$ in the path incurs an increase of $\vec{j} - \vec{i}$ in the value of $\vec{\beta}(P^*)$. In graph $G^\dagger$, graph structure replaces the attribute information except for β_5, which annotates the edges of $G^\dagger$ in the same way as edges of G^*. For example, consider cut $C_4 = \{ce, bd\}$ or equivalently $P_4^* = \{\text{dash, black}\}$. Using graph G^* [Fig. 8(a)], we calculate $\vec{\beta}(P_4^*)$ as $[2, 2, 4, 4, 1] + [2, 1, 4, 4, 1] = [4, 3, 8, 8, 2]$, while in graph $G^\dagger$ we have the same value (the first four elements of the vector) by simply looking up the index of t^* at the end of path P_4^* [Fig. 8(d)].

Therefore, one can evaluate (4) for all possible paths by enumerating the vertices t^* at the end of every path P^* in graph $G^\dagger$. Let us denote the end point as $\tilde{t}(P^*)$. For example, $\tilde{t}(P_4^*) = [4, 3, 8, 8]$, because P_4^* ends at vertex $t^*[4, 3, 8, 8]$. The problem can be reformulated as

$$\begin{aligned} &\text{a) minimize:} \quad Q(P^*) = F\big(\tilde{t}_1(P^*), \tilde{t}_2(P^*), \beta_5(P^*)\big) \\ &\text{b) constraints: } \tilde{t}_3(P^*) \le \beta_3^{\max} \text{ and } \tilde{t}_4(P^*) \le \beta_4^{\max}. \end{aligned} \tag{5}$$

Note that constructing $G^\dagger$ does not reduce algorithmic complexity of the partitioning. It is merely outlined for better visualization, and to enable subsequent optimizations.

C. Hard Constraints

A path in $G^\dagger$ that visits the vertex $v[\vec{i}]$ will lead to a partitioning solution whose first four attributes are at least as large as the elements of i. Therefore, we can trim out infeasible solutions, violating at least one of the hard constraints, during construction of $G^\dagger$. This is achieved not by trimming paths that violate (5.b). after construction of $G^\dagger$, but by refusing to insert violating edges in $G^\dagger$ in the first place. Each time we are about to connect an existing vertex $u^*[\vec{i}]$ to a new vertex $v^*[\vec{j}]$, violations of the hard constraints can be detected, in which case, vertex v and the edge are discarded. For example, in Fig. 8, assume $\beta_3^{\max} = 10$ and $\beta_4^{\max} = 11$. Path P_1^* violates the constraint because $0+\beta_4(ab) \ge \beta_4^{\max}$, i.e., the fourth attribute of the new vertex $t^*[1, 5, 2, \underline{13}]$ is beyond the limit. P_6^* violates the constraint too. Thus, vertex $u^*[\vec{i}]$ is connected to vertex $v^*[\vec{j}]$ only if

$$\begin{cases} \text{there exists an edge } e^*(u^*, v^*) \text{ in } G^* \\ \text{and } \vec{i} + \vec{\beta}(e^*) = \vec{j} \\ \text{and } j_3 \le \beta_3^{\max} \text{ and } j_4 \le \beta_4^{\max}. \end{cases}$$

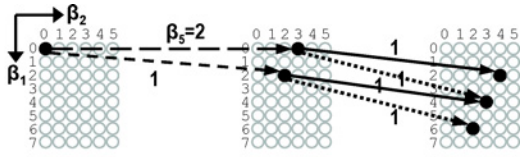
As a result, graph $G^\dagger$ is constructed such that any path from $s*$ to a t^* vertex in $G^\dagger$ identifies a feasible solution. It follows that (5.b). can be removed from consideration, and fewer paths that are left in $G^\dagger$ should be evaluated to optimize (5.a).

Other constraints can be encoded similarly. For example, one could embed hard constraints on the maximum workload that could be assigned to either processor, which translates to trimming edges in $G^\dagger$ based on the first or second attribute.

D. Cost Function Minimization

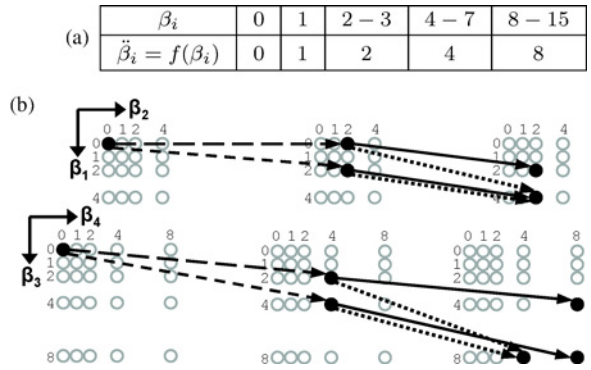
The first four elements in the attribute vector of a path from $s^*[\vec{0}]$ to $\tilde{t}$ can be expressed based on the coordinates of the $\tilde{t}$ vertex in 4-D space. Hence, all paths that terminate at the same end point in $G^\dagger$ share the same attribute vector. That is, all such paths lead to task assignments that incur the same workload and memory distribution, and are only different in their inter-processor communication volume.

Among such possible task assignments, one would typically prefer the solution that incurs the minimum communication. This is to say that the cost function Q is non-descending in the fifth element of the attribute vector, $\beta_5(P^*)$. Equivalently, among a group of competing paths terminating at the same end point, we are interested in the path with minimum β_5.



t_1, t_2	shortest path for this t	Q = max(W_1+N, N+W_2)	best path
2,4		max(2+3, 3+4) = 7	
4,3		max(4+2, 2+3) = 6	✓
6,2		max(6+2, 2+2) = 8	

Fig. 9. Finding the shortest paths to t^* end points, and evaluating the cost function $Q = \max\{W_1 + N, W_2 + N\}$ for them.



	β_i	0	1	2 − 3	4 − 7	8 − 15
(a)	$\ddot{\beta}_i = f(\beta_i)$	0	1	2	4	8

Fig. 10. (a) Approximation function. (b) Constructing the approximated graph $G^\dagger$ from Fig. 8.

Recall that the edges of $G^\dagger$ are annotated with their β_5 attribute. We can treat edge annotations as distance labels in $G^\dagger$, and run single-source multiple-destination shortest path on the graph from source vertex $s^*[\vec{0}]$ to all possible $\tilde{t}$ end points. The shortest path procedure would prune many possible paths that do not minimize cost function Q. Specifically, for every possible end point $\tilde{t}$, it only leaves one path in the graph that terminates there. For example, in Fig. 8(d), two paths arrive at the same end point: $\tilde{t}(P_3^*) = \tilde{t}(P_4^*) = [4, 3, 8, 8]$. However, only P_4^* is maintained for end point $t^*[4, 3, 8, 8]$ (Fig. 9).

We can calculate the value of Q at each of the $\tilde{t}$ points to find the end point, and the corresponding path, which globally minimizes Q (Fig. 9). This procedure delivers an exact solution, which is guaranteed to optimize the cost function subject to hard constraints.

One might be able to prune out more candidate points from consideration, if more specific information about the cost function is known. For example, if one wants to avoid highly skewed workload distribution, he can ignore the end points whose w_1 or w_2 attribute is below or above a certain threshold.

E. Algorithm Complexity

The task graph and subsequent graphs constructed from that are directed acyclic graphs. Hence, the complexity of discussed transformations and the shortest path algorithm grow linearly with the number of edges in the subject graphs. Note that on a DAG, single source shortest path can be implemented using topological sort and thus has linear complexity [19].

In planar graphs, the number of edges grows linearly with the number of vertices. Therefore, time complexity of our algorithm is determined by the number of vertices in the largest subject graph, i.e., $G^\dagger$. There are at most $N \times \beta_1^{\max} \times \beta_2^{\max} \times \beta_3^{\max} \times \beta_4^{\max}$ vertices in $G^\dagger$, where N is the number of vertices in the application task graph. Thus, our algorithm has the time complexity of $O(N \times \prod_{i=1}^4 \beta_i^{\max})$.

This is considered pseudo-polynomial because the terms $\beta_i^{\max}$ merely depend on the properties of the application graph and target platform. $\beta_3^{\max}$ and $\beta_4^{\max}$ represent the available memory on the processors. Moreover, $\beta_1^{\max}$ is independent of the path choice, because $\beta_1(P^*) = \beta_1(C) = \sum_{v \in G_1} w_1(v)$ and thus its maximum value, $\beta_1^{\max}$, is the computation workload of the entire application when all tasks are assigned to processor 1. A similar argument holds for $\beta_2^{\max}$.

Note that since our graph bi-partitioning problem is NP-Complete no algorithm with strictly-polynomial complexity exists unless $P = NP$. The pseudo-polynomial complexity does not impose a real constraint on practicality of our approach. Pseudo-polynomial time algorithms incur reasonable latency for typical problem instances, unless we have to deal with very large attribute numbers which are uncommon in many practical settings [20]. In practice, the attributes can be normalized so that $\beta_i^{\max}$ values remain relatively small.

VIII. Approximation Method

In this section, we present an approximation method for our exact task assignment algorithm with strictly-polynomial complexity. The approximation algorithm takes as input an acceptable error bound ξ, e.g., 10%, and guarantees that quality, i.e., throughput, of the near-optimal solution would not be more than a factor of ξ away from the optimal throughput.

We reduce the complexity by simplifying graph $G^\dagger$ in the exact algorithm. We illustrate that partitioning quality is not degraded beyond the intended limit in the course of the process. Specifically, the number of vertices in the expanded graph $G^\dagger$ is reduced to $O(N \times \prod_{i=1}^4 \log \beta_i^{\max})$ from the original $O(N \times \prod_{i=1}^4 \beta_i^{\max})$. In each of the four β_i dimensions of the 4-D space in $G^\dagger$, we judiciously trim the $\beta_i^{\max}$ possible values of indices to only $\log \beta_i^{\max}$ distinct numbers.

The idea is to replace multiple attribute values with a single representative value for that attribute. That is, we would like to develop an approximation function $\ddot{\beta} = f(\beta)$ to generate a representative value $\ddot{\beta}$ for a range of β values. For a given number $\delta > 0$, the approximation function is defined as

$$\ddot{\beta} = f(\beta) = (1 + \delta)^{\left\lfloor \log_{1+\delta}^{\beta} \right\rfloor} \quad \text{and} \quad f(0) = 0.$$

We apply this function whenever a new edge is to be added to graph $G^\dagger$. Thus, for constructing the path corresponding to P^* from s^* to t^* in G^*, the approximation function is applied k times, where k is the number of edges in path P^*. Fig. 10 illustrates the idea when $\delta = \vec{1}$, and possible values are trimmed to only 0, 1, 2, 4, and 8.

For example, consider path $P_3^* = \{\text{longdash, dot}\}$ in Fig. 10. Originally, the longdash-marked edge connected $s^*[0, 0, 0, 0]$

to $r^*[0+0, 0+3, 0+2, 0+7] = r^*[0, 3, 2, 7]$. After applying the approximation, the path ends at $r^*[f(0+0), f(0+3), f(0+2), f(0+7)] = r^*[0, 2, 2, 4]$. Similar procedure is applied to the next edge in the path, i.e., the dot-marked edge. It starts from the approximated vertex $r^*[0, 2, 2, 4]$ and ends in $t^*[f(0+4), f(2+0), f(2+6), f(4+1)] = t^*[4, 2, 8, 4]$.

After approximation, the number of vertices in the graph $G^\dagger$ becomes proportional to $N \times \Pi_{i=1}^{4} \log \beta_i^{\max}$, because in each of the four β_i dimensions, the above approximation function results in one of the following possible distinct numbers: $0, 1, 1+\delta, (1+\delta)^2, \cdots, (1+\delta)^b$, where $b = \left\lfloor \log_{1+\delta}^{\beta_i^{\max}} \right\rfloor$.

Lemma 1: We have $\dfrac{\beta}{1+\delta} < \ddot{\beta} \leq \beta$.

Theorem 3: If we set $\delta = \sqrt[F]{1+\epsilon} - 1$, where $\epsilon > 0$ and F is the number of faces in task graph G, then we have

$$\frac{\beta(P^*)}{1+\epsilon} < \ddot{\beta}(P^*) \leq \beta(P^*).$$

This means that for every path P^* from s^* to t^* in the expanded graph $G^\dagger$, the approximated value $\ddot{\beta}(P^*)$ is at most degraded by a factor of $1+\epsilon$ from its original value $\beta(P^*)$.

Note that β and $\ddot{\beta}$ are vectors. In fact, δ is also a vector with four elements, which implies that we can have different approximation factors for different dimensions of the 4-D space. All mathematical operations are performed on each of the four dimensions independently.

The following theorems quantify the impact of approximation on memory constraints and cost function minimization.

Corollary 2: The original hard constraints

$$\beta_3(P^*) \leq \beta_3^{\max} \quad \text{and} \quad \beta_4(P^*) \leq \beta_4^{\max}$$

can be replaced with the following constraints, which use approximated values

$$\ddot{\beta}_3(P^*) \leq \frac{\beta_3^{\max}}{1+\epsilon} \quad \text{and} \quad \ddot{\beta}_4(P^*) \leq \frac{\beta_4^{\max}}{1+\epsilon}.$$

Corollary 3: Let $\ddot{\beta}^{\max} = f(\beta^{\max})$. The constraints

$$\ddot{\beta}_3(P^*) \leq \ddot{\beta}_3^{\max} \quad \text{and} \quad \ddot{\beta}_4(P^*) \leq \ddot{\beta}_4^{\max}$$

guarantee that

$$\beta_3(P^*) \leq (1+\epsilon)\beta_3^{\max} \quad \text{and} \quad \beta_4(P^*) \leq (1+\epsilon)\beta_4^{\max}.$$

Therefore, to construct the new set of memory constraints, we may use either of the above two corollaries. The formula in Corollary 2 guarantees that the original constraints are met, at the expense of trimming some possibly-valid paths. The formula in Corollary 3 provides a new constant bound within a factor of $1+\epsilon$ from the bound in the original constraint. We proceed to consider the impact of approximation on cost function minimization.

Theorem 4: Let $\ddot{Q}(P^*) = F\big(\ddot{\beta}_1(P^*), \ddot{\beta}_2(P^*), \beta_5(P^*)\big)$ denote the approximated value of our cost function $Q(P^*) = F\big(\beta_1(P^*), \beta_2(P^*), \beta_5(P^*)\big)$, for the path P^*. We have

$$\Big(1 - \frac{\epsilon}{1+\epsilon} S(P^*)\Big)\ Q(P^*) \ \leq\ \ddot{Q}(P^*) \ \leq\ Q(P^*)$$

where $S(P^*)$ is defined as

$$S(P^*) = \frac{\beta_1(P^*)}{Q(P^*)} \max \frac{\partial Q}{\partial \beta_1} + \frac{\beta_2(P^*)}{Q(P^*)} \max \frac{\partial Q}{\partial \beta_2}.$$

Note that although Q is originally a discrete function, throughout the following mathematical analysis, we look at it as a continuous function. That is, we use the same formula for Q, but assume its domain is $\mathbb{R}$ instead of $\mathbb{I}$. Note that Q does not have to be differentiable. As long as Q is differentiable on several intervals and continuous (i.e., piece-wise differentiable), we are able to calculate the maximum slope. For example, $\max \frac{\partial Q}{\partial \beta_1} = \max \frac{\partial Q}{\partial \beta_1} = 1$ for $Q = \max\{W_1 + \alpha_1 N, \alpha_2 N + W_2\} = \max\{\beta_1 + \alpha_1 \beta_5, \alpha_2 \beta_5 + \beta_2\}$, because the slope of Q with respect to both β_1 and β_2 is either 0 or 1 on its entire domain.

Corollary 4: Let $S^{\max}$ be the maximum possible value of S over the domain of function Q. We have

$$\forall P^* : \Big(1 - \frac{\epsilon}{1+\epsilon} S^{\max}\Big)\ Q(P^*) \leq \ddot{Q}(P^*) \leq Q(P^*).$$

The above theorem states that the error in calculating cost function is bounded within a constant factor. The main objective of graph bi-partitioning is to find the optimal path $P^* = P^*_{\text{opt}}$ which minimizes our cost function $Q(P^*)$. Using the approximation method, however, $\ddot{Q}(P^*)$ is minimized for some near-optimum path $P^* = P^*_{\text{near}}$.

Corollary 5: Let $\xi = \dfrac{\epsilon}{1+\epsilon} S^{\max}$, and $T = \dfrac{1}{Q}$ denote the throughput. We have

$$(1 - \xi)\ T_{\text{opt}} \leq T_{\text{near}} \leq T_{\text{opt}}.$$

Consequently, for a given tolerable error bound ξ, throughput of the near-optimal solution would not be more than a factor of ξ away from the optimal throughput. The appropriate approximation factor δ is calculated using ξ in the following manner. First, we calculate ϵ from equation $\xi = \frac{\epsilon}{1+\epsilon} S^{\max}$, and then, δ from equation $\delta = \sqrt[F]{1+\epsilon} - 1$. The approximation process provides a controllable knob to trade task assignment quality with computation complexity and runtime.

IX. Hardware-Inspired Versatility and Extensibility

We presented our technique using a selected set of attributes and an unconstrained cost function. In this section, we discuss extensions to our proposed task assignment algorithm, and demonstrate its utility in handling a diverse set of hardware customization scenarios.

One aspect of versatility is the ability to model execution period as a general, rather arbitrary, function of processor workloads (W_i's) and communication traffic (N). This provides a simple yet effective way to fine tune the process of software synthesis to better match a specific hardware architecture. As an example, consider the hardware in Fig. 11, and assume that there is no cache. A reasonable choice of the cost function would be $Q = \max\{\frac{W_1+\alpha_1 N}{150}, \frac{hop}{400}, \frac{W_2+\alpha_2 N}{200}\}$, to account for the disparity in clock frequency, network bandwidth, and its latency. The term *hop* accounts for the router speed, and varies based on router implementation and statistical

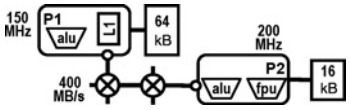


Fig. 11. Sample heterogeneous dual processor hardware.

patterns of the traffic from other possible units. This cost function accounts for the details of onchip network, as well as the heterogeneity in processors' micro-architecture and clock frequency.

The technique can also be extended to handle other attributes. For example, in Fig. 11, using the w_i and n attributes alone, it is not easy to account for the effect of L1 cache on processor 1. In this case, we can add an additional attribute $ls_1(v)$ to vertices of task graph G. The attribute $ls_1(v)$ models the dynamic count of load/store instructions, in one firing of the task corresponding to vertex v. Similar to W_1, we can define $LS_1 = \sum_{v \in V_1} ls_1(v)$. Subsequently, execution period estimation cost function can be generalized to $Q = F(W_1, W_2, N, LS_1)$, in order to consider the impact of the new attribute on application throughput. For example, one might use the function $Q = \max\{\frac{LS_1 \times mr \times mp + W_1 + \alpha_1 N}{150}, \frac{hop}{400}, \frac{W_2 + \alpha_2 N}{200}\}$ to model the impact of cache misses. The extra cycles in processor 1 are equal to $LS_1 \times mr \times mp$, where mr and mp are the approximated miss rate and miss penalty, respectively. Note that we compact all attributes into one vector. As a result, addition of a new attribute, such as $ls_1(v)$, adds a dimension to the attribute vector and the space of graph $G^\dagger$.

Similarly, we can remove extra attributes and simplify the formulation in case they are not relevant for a given hardware. For example, if processors are homogeneous, we may eliminate attribute $w_2(v)$ and replace the term W_2 in the performance estimation function with $W - W_1$, where $W = \sum_{v \in G} w_1(v)$. In such cases, $\vec{\beta}$ would have fewer elements, and graph $G^\dagger$ would be constructed in a space with fewer dimensions.

We may also introduce or eliminate hardware-inspired hard constraints. For example, if we have a target hardware with separate data and instruction memory modules, we can introduce distinct constraints on the size of each memory module, e.g., $D_1 \leq$ data memory of processor 1, and $I_1 \leq$ instruction memory of processor 1. Similarly, one could introduce hard constraints on the processor workloads to eliminate solutions that result in highly skewed workload distributions.

X. Empirical Evaluation

A. Soft Dual-Processor Platforms

Our evaluation is based on measurements of application throughput and memory requirement using operational binary on prototyped hardware. We use Altera DE2 FPGA board to implement soft dual-processor platforms using NiosII/f cores [21]. NiosII/f is a 32-bit, in-order, single issue, pipeline reduced instruction set computer processor with configurable architectural parameters.

We experiment with several different hardware configurations. In all cases, processors and the communication link run at 100 MHz. Processors use tightly coupled instruction memory constructed out of FPGA's onchip random-access memory blocks. Both processors have integer multipliers, but in $P1$ it is built out of logic elements (LEs), and in $P2$ it is built out of FPGA's embedded digital signal processing blocks, which offers better performance. Fig. 12 summarizes the configuration of the $2 \times 2 \times 3 = 12$ soft dual-processor platforms used in our paper.

Name	Processor P1			Link	Processor P2		
	int	float	imem		int	float	imem
A-F-08-24	LE	-	8K	fifo	DSP	FPU	24K
A-F-16-16			16K				16K
A-F-24-08			24K				8K
B-F-08-24	LE	FPU	8K	fifo	DSP	-	24K
B-F-16-16			16K				16K
B-F-24-08			24K				8K
A-R-08-24	LE	-	8K	router	DSP	FPU	24K
A-R-16-16			16K				16K
A-R-24-08			24K				8K
B-R-08-24	LE	FPU	8K	router	DSP	-	24K
B-R-16-16			16K				16K
B-R-24-08			24K				8K

Fig. 12. Target soft dual-processor platforms.

1) *Computation:* We experiment with two microarchitecture configurations, referred to as `A` and `B`. In setting `A`, $P2$ has a hardware floating point unit (FPU), while $P1$ uses software emulation to carry out floating point operation. In `B`, $P1$ has FPU and $P2$ uses software emulation.

2) *Communication:* Similarly, there are two different communication links (`F` or `R`). In setting `F`, inter-processor link is a 32-bit FIFO channel with 256 words buffer. We instantiate the link using Altera Onchip FIFO Memory in System on a Programmable Chip Builder software. In setting `R`, processors are connected through a third processor that emulates the function of a router in a network-on-chip (NoC). To model the impact of network traffic on message delivery latency in NoC, routing latency is assumed to be a random variable with normal distribution.

3) *Memory:* A total of 32 kB is available for instruction memory. We consider three different settings for allocating memory to processors: `08-24`, `16-16`, and `24-08`. For example, the first row in Fig. 12 refers to a configuration in which processors $P1$ and $P2$ have 8 kB and 24 kB of memory, respectively. The next two rows show two different distributions of memory space between the two processors.

B. Evaluation Methodology

We implemented our method in StreamIt 2.1 compilation framework [6]. The compiler takes as input an application specified in StreamIt language, and after static scheduling and partitioning of the graph, generates separate `C` codes for execution on parallel processors. To generate executable binary, each `C` code is compiled with NiosII IDE C compiler (-O2 optimization) for its corresponding target Nios processor. Subsequently, applications' code sizes are obtained from the generated executable binaries. If the code size is larger than the available instruction memory, it is not possible to execute it on the soft processors. For feasible cases, the binaries

are mapped to corresponding processors in prototyped architectures. Subsequently, application steady state throughput is measured during execution on FPGA.

We compare three different task assignment algorithms. First, we evaluate our proposed versatile algorithm described in this paper, including our approximation method with parameter $\epsilon = 0.1$ for workload values. We also experiment with our recent task assignment method [12], [13], which is provably-optimal for homogeneous processors. Third, we use StreamIt built-in task assignment algorithm, which does not directly address heterogeneous architectures or memory constraints [6]. We refer to our proposed versatile algorithm as `Ver`, our previous work as `Pre`, and StreamIt as `Str`.

`Pre` handles heterogeneous architectures only by estimating their relative performance with a constant factor r, which is established by profiling the processors using a set of representative applications. For example, we found that for AF and AR configurations, $r = 0.7$ models the relative performance of processor $P1$ over $P2$, for benchmark applications containing floating point operations. Similarly, $r_{AF} = r_{AR} = 0.4$ is the relative performance for integer benchmarks. *Pre* does not consider memory constraints.

C. Attribute Estimation

Using high-level information available to the compiler, we estimate the attributes with simple models. Our empirical observations validate the effectiveness of the models.

1) *Communication:* Our adopted application model explicitly specifies the amount of data transfered between two communicating tasks. For example, in StreamIt synchronous dataflow graphs, each task v fires a statically-known number of times $s(v)$ in the periodic steady state schedule [22]. The number of data samples produced and consumed per firing of a task is also explicitly specified. Let $p(vu)$ denote the number of data samples sent from v to u, every time v fires. It follows that $n(vu) = s(v) \times p(vu)$.

2) *Computation:* We profiled NiosII/f processors to estimate their cycle per instruction distribution. Subsequently, internal computations of tasks are analyzed at high-level, and a rough mapping between high-level StreamIt language constructs and processor instructions is determined. For SDF-compliant streaming applications, control-flow characteristics are minimal. As a result, we employed first order estimations, such as average if-then-else path latencies, whenever needed. The analysis derived $w_i^*(v)$, which represents clock cycles needed for one firing of vertex v on processor i. Therefore, $w_i(v) = s(v) \times w_i^*(v)$.

3) *Memory:* Estimating memory requirement is similar to the above workload estimation, except that here we count the number of assembly instructions corresponding to high-level statements of StreamIt language without actually compiling the applications to assembly. The analysis derived $m_i^*(v)$, which represents the amount of instructions memory needed for one firing of vertex v on processor i. Therefore, $m_i(v) = \text{mem}(s(v), for) + m_i^*(v)$, where $\text{mem}(s(v), for)$ is for the loop that would iteratively execute task v. Hence, if $s(v) = 1$, $\text{mem}(s(v), for) = 0$. Otherwise, it is a constant number, e.g., 12 in our platforms. Note that estimations are tuned to -O2 compiler optimization switch.

Application	Description	Operations	$\|V\|$	$\|E\|$
`FFT`	Fast Fourier Transform	float	81	105
`TDE`	Time Domain Equalizer	float	50	60
`MMF`	Blocked Matrix Multiply	float	21	21
`MMI`	Blocked Matrix Multiply	int	21	21
`SORT`	Bitonic Sort	int (no mult.)	314	407

Fig. 13. Benchmark applications. The last two columns show the number of vertices and edges in the application task graph.

	Processor	`FFT`	`TDE`	`MMF`	`MMI`	`SORT`
Estimate (bytes)	P1	2828	3944	3260	1920	7644
	P2	14 384	13548	3740	2992	22 832
Actual (bytes)	P1	2844	3864	3340	2320	5908
	P2	14 560	12 280	4196	3036	21 296

Fig. 14. Accuracy of high-level memory estimates for `A-F-08-24`.

D. Performance Estimation and Memory Constraints

We mentioned that the cost function modeling application execution period should be tailored to target platform. In the first six target configurations, `A-F-` and `B-F-`, the communication link is a FIFO channel with single-cycle delivery latency. Therefore, we customize the cost function $Q_{AF} = Q_{BF} = max(W_1 + \alpha_1 N, W_2 + \alpha_2 N)$, with $\alpha 1 = \alpha_2 = 8$. The parameter $\alpha 1$ (α_2) models the additional workload that has to be introduced on the sender (receiver) to write (read) a word of data to (from) the FIFO. Its constant value depends on our software generation setup. Similarly, in the next six targets, `A-R-` and `B-R-`, we use $Q_{AR} = Q_{BR} = max(W_1 + 8N, N \times avg(L), W_2 + 8N)$, where $avg(L)$ is the average value of the random latency in the router.

In all cases, hard constraints are introduced to satisfy instruction memory constraints. There is no constraint on data memory, since it is large enough for all our benchmark applications. Hence, we only maintain attributes $m_i(v)$ [and not $m_i(e)$] because they represent code sizes. In case of `A-F-08-24`, for example, the set of constraints is $M_1(V_1) \leq 8$ kB and $M_2(V_2) \leq 24$ kB.

E. Benchmarks

Fig. 13 shows different benchmarks used in our evaluations. They are well-known streaming applications that frequently appear in embedded application space. The applications are selected from the StreamIt benchmark set [6], considering the memory constraints of our FPGA board.

F. Results

Fig. 14 shows the accuracy of our high-level code size estimation scheme. The comparison and estimation accuracy are significant, because our task assignment algorithm uses the code size information to guarantee that generated binary code will fit in the limited instruction memory of soft processors. Note that we decided to use estimates because at the time of task assignment the binary code is not available and is

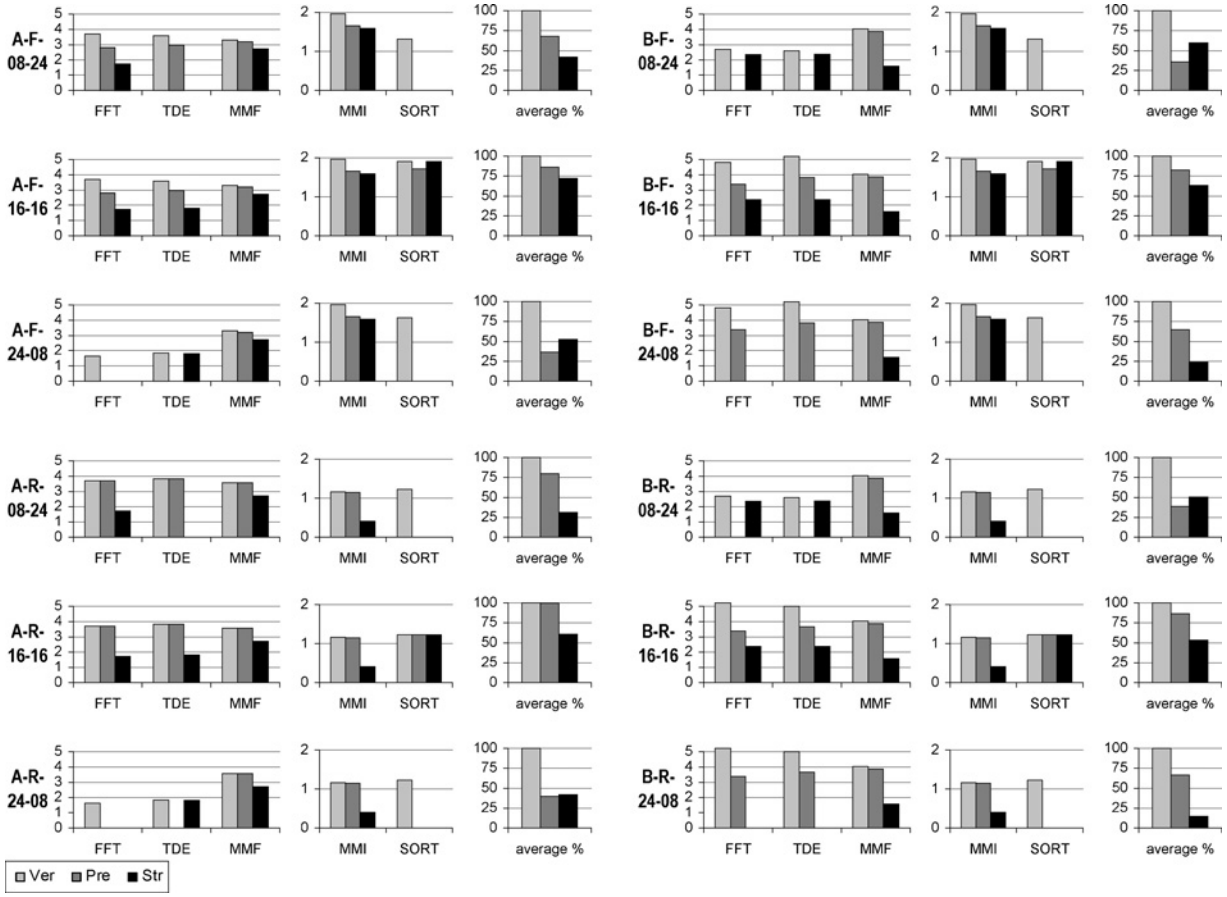


Fig. 15. Applications' throughput normalized with respect to a processor with LE multiplier and no FPU. `Pre` and `Str` failed to generate feasible implementations for some architectures.

yet to be compiled. In addition, tasks are not *accurately* compile-able in isolation either, because of a variety of reasons such as volume-dependent communication, shared variable definitions, and the difference between inter-processor versus intra-processor communication code.

On average, over all configurations and benchmarks, our high-level code size estimates are within 10% of the size of compiled binary. Some inaccuracy is inevitable at high-level, due to lack of information about many compilation and optimization decisions in generating binary code.

Fig. 15 presents measured application throughput, normalized with respect to a single processor with a LE multiplier and without an FPU. The experiments highlight that the versatile algorithm (`Ver`) always produces code that fits in limited instruction memory of soft processors. While for `-08-24` and `-24-08` configurations, the other two algorithms (`Pre` and `Str`) fail to generate feasible implementations for some benchmarks. For example, in case of `SORT` application, which needs about 32 kB memory, they always fail. `MMI` and `MMF` require small amount of memory, and therefore, they are feasibly implemented with all the three algorithms.

In addition, `Ver` consistently outperforms the other two techniques, and delivers the highest throughput in all cases. This underscores versatility of our proposed technique, and the fact that it can effectively handle heterogeneity and specific requirements of different target architectures.

Both `Pre` and `Ver` algorithms do not explicitly consider the impact of task assignment on code size, and therefore, fail to generate feasible solutions in all cases. For architectures with `-16-16` suffix, which have a balanced distribution of memory, all three algorithms produce feasible implementations.

`SORT` application contains a sequence of integer comparisons and conditional swaps, and no multiplications or floating point operations. Thus, `SORT` is indifferent to the micro-architecture heterogeneity that exists in our platforms. That is why `Str`, which assumes homogeneous processors, delivers the same throughput as `Ver`. On the other hand, `Pre` considers an average 0.7 factor in relative performance, which is irrelevant for `SORT`, and hence, degrades throughput.

Hardware FPU is substantially more efficient than software emulation of floating point operations. `MMF`, `TDE`, and `FFT` have large numbers of such operations, and hence, their best throughput is observed when we assign as many tasks as possible to the processor with hardware FPU. Our integrated memory size estimation and heterogeneity modeling enables us to considerably outperform the competitors for floating point applications. In general, considering the effect of heterogeneous architecture is essential for most of the applications. Here, we can clearly see the advantage of `Ver` over the other two algorithms. `Pre` only partially considers platforms' heterogeneity, and often outperforms `Str`.

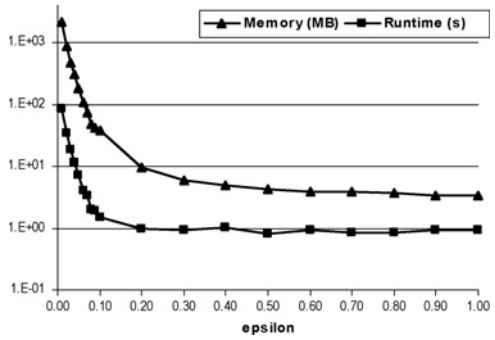


Fig. 16. Memory usage and execution time of `Ver` for different approximation accuracy factors. It is measured for `FFT` on `A-F-100-100` platform.

Fig. 16 shows the tradeoff in execution time and memory usage of `Ver`, measured on a Linux/P4 machine for `FFT` benchmark and `A-F-100-100` platform, with respect to approximation accuracy. As expected, both runtime and memory usage of the algorithm decrease with decrease in approximation accuracy, i.e., with larger ϵ. Effectively, ϵ gives users a controllable knob by which, they can optimize algorithm's runtime and memory footprint in return for willingness to accept small deviation from optimal solution. For example, in case of Fig. 16, one could save about two orders of magnitude in runtime and memory usage by accepting at most 20% loss in quality, relative to the exact solution.

XI. Conclusion and Future Directions

We have presented a versatile task assignment algorithm that both maximizes the steady state throughput and meets the possible memory size constraints. We experimented with the algorithm on different heterogeneous hardware configurations and measured the throughput and code sizes. As we saw in the experiments, our algorithm always produced code that fits in the limited memory resources, and also, it has the best throughput comparing with two recent competitors. Future directions include studying compilation frameworks that not only support heterogeneous processors but also optimize the design for multiple processors.

References

[1] M. Butts, A. DeHon, and S. C. Goldstein, "Molecular electronics: Devices, systems and tools for gigagate, gigabit chips," in *Proc. Int. Conf. Comput.-Aided Design*, 2002, pp. 433–440.

[2] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2006, pp. 201–210.

[3] Y. Jin, N. R. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synthesis*, 2005, pp. 273–278.

[4] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2004, pp. 183–189.

[5] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. Int. Conf. Compiler Construction*, 2002, pp. 179–196.

[6] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *Proc. Int. Conf. Architectural Support Programming Languages Operating Syst.*, 2002, pp. 291–303.

[7] W. Thies, J. Lin, and S. Amarasinghe, "Partitioning a structured stream graph using dynamic programming," in *Proc. 5th Workshop Media Streaming Processors*, Dec. 2003.

[8] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. Int. Conf. Architectural Support Programming Languages Operating Syst.*, 2006, pp. 151–162.

[9] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for DSP using Ptolemy," *J. Very Large Scale Integr. Signal Process. Syst.*, vol. 9, nos. 1–2, pp. 7–21, Jan. 1995.

[10] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proc. Design Autom. Conf.*, Jun. 2007, pp. 777–782.

[11] J. Cong, G. Han, and W. Jiang, "Synthesis of an application-specific soft multiprocessor system," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2007, pp. 99–107.

[12] M. Hashemi and S. Ghiasi, "Exact and approximate task assignment algorithms for pipelined software synthesis," in *Proc. Conf. Design Autom. Test Europe*, 2008, pp. 746–751.

[13] M. Hashemi and S. Ghiasi, "Throughput-driven synthesis of embedded software for pipelined execution on multi-core architectures," *ACM Trans. Embedded Comput. Syst.*, vol. 8, no. 2, article no. 11, Jan. 2009.

[14] E. A. Lee, "The problem with threads," *IEEE Comput.*, vol. 39, no. 5, pp. 33–42, May 2006.

[15] A. D. Pimentel, L. O. Hertzbetger, P. Lieverse, P. van der Wolf, and E. E. Deprettere, "Exploring embedded-systems architectures with Artemis," *IEEE Comput.*, vol. 34, no. 11, pp. 57–63, Nov. 2001.

[16] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Comput.*, vol. 36, no. 4, pp. 45–52, Apr. 2003.

[17] G. Kahn, "The semantics of simple language for parallel programming," in *Proc. Int. Federation Inform. Process. (IFIP) Congr.*, 1974, pp. 471–475.

[18] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Single sourse shortest path in directed acyclic graphs," in *Introduction to Algorithms*. Cambridge, MA: MIT Press and McGraw-Hill, 2001, pp. 592–595.

[20] M. R. Garey and D. S. Johnson, "Number problems and strong NP completeness," in *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman, 1990, pp. 90–92.

[21] *Altera Nios* [Online]. Available: www.altera.com/nios

[22] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.

Matin Hashemi (S'10) received the B.S. degree in electrical engineering from the Sharif University of Technology, Tehran, Iran, in 2005, and the M.S. degree in computer engineering from the Department of Electrical and Computer Engineering, University of California, Davis, in 2008, where he is currently working toward the Ph.D. degree in computer engineering. His research interests include the area of embedded systems.

Soheil Ghiasi (S'98M'05) received the B.S. degree in computer engineering from the Sharif University of Technology, Tehran, Iran, in 1998, and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles (UCLA), in 2002 and 2004, respectively.

He has been with the University of California, Davis, since he graduated from UCLA in 2004, where he is currently an Assistant Professor of Electrical and Computer Engineering with the Department of Electrical and Computer Engineering. His research interests include design and optimization of embedded computing systems.

Dr. Ghiasi received the Harry M. Showman Prize for excellence in research communication from the College of Engineering, UCLA, in 2004.