Throughput-Driven Synthesis of Embedded Software for Pipelined Execution on Multicore Architectures

MATIN HASHEMI and SOHEIL GHIASI
University of California, Davis

11

We present a methodology for pipelined software synthesis of streaming applications. First, we develop a versatile task assignment algorithm capable of optimizing realistically-arbitrary cost functions for two cores. The algorithm is exact (i.e., theoretically optimal) contrary to existing heuristics. Second, our approximation technique provides an adjustable knob to trade solution quality with algorithm runtime and memory. Third, we develop a recursive heuristic for more cores. FPGA-based emulated experiments validate our theoretical results. The exact algorithm yields 1.7× throughput improvement. The approximation method offers a range of tradeoff points (e.g., 3× faster with 20× less memory) while degrading the throughput only 1% to 5%.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; D.3.4 [**Programming Languages**]: Processors–Optimization

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Embedded software, graph partitioning, multi-core hardware, streaming applications, task assignment

ACM Reference Format:

Hashemi, M. and Ghiasi, S. 2009. Throughput-driven synthesis of embedded software for pipelined execution on multi-core architectures. ACM Trans. Embedd. Comput. Syst. 8, 2, Article 11 (January 2009), 35 pages. DOI = 10.1145/1457255.1457258 http://doi.acm.org/10.1145/1457255.1457258

1. INTRODUCTION

Despite the tremendous societal and economical potential of cyber-physical systems, the process of application development for such systems is largely ad hoc today [Sztipanovits et al. 2005; Lee 2006; Henzinger and Sifakis 2006]. Presently, the practitioners have to settle for slow and costly development procedures, which yield unreliable and unportable software [Lee 2005]. Many researchers are working to develop a formal *science* to pave the way to systematic

Authors' addresses: M. Hashemi and S. Ghiasi, University of California, Davis; email: hashemi@ucdavis.edu.

and high-confidence development of embedded systems [Sztipanovits et al. 2005; Meeting 2006].

In order to automatically realize an application software, a sequence of refinement stages have to be carried out to bridge the gap between high-level specifications and application implementation. Formal modeling and model checking, such as existing results on interfacing inherent continuity of physical world with discreteness of digital computing [Alur et al. 1992; Henzinger et al. 1992; Stankovic 2007] and guaranteeing properties of computing processes [Henzinger et al. 1994; Henzinger et al. 1998; Alur 2003], provide means to ensure quality and correctness of specifications. Refinement stages of the synthesis need to preserve such properties and deliver provably efficient implementations [Bonivento et al. 2005; Benveniste et al. 2003; Pinto et al. 2006].

This article takes a step toward the grand challenge of bridging the gap between formal application specification and its implementation. We study the problem of task assignment during synthesis of embedded streaming applications on parallel processors, where applications are modeled as task graphs or synchronous dataflow [Lee and Messerschmitt 1987b].

Streaming applications are characterized by the requirement to process a virtually infinite sequence of data items under performance constraints [Owens et al. 2002; Thies et al. 2002]. They are becoming increasingly important and widespread, and they appear in many disciplines such as networking, signal processing, security, and multimedia. Typically, they demand high throughput but are not very sensitive to response latency. Hence, pipelined execution is a favorable design choice for their implementation [Gordon et al. 2002; Rangan et al. 2006; Owens et al. 2000].

The contribution of this article is threefold. First, we present a task assignment methodology for synthesizing pipelined streaming applications that execute on dual-core platforms. The objective is to maximize the pipeline throughput. Our proposed method is an exact algorithm, as opposed to many existing heuristic approaches that do not provide a guarantee on quality of task assignment.

In addition to being optimal (i.e., exact), our technique is also versatile, meaning that it can readily optimize any realistically[1] arbitrary function of computation workloads and interprocessor communications. Therefore, our task assignment supports different situations, such as heterogeneous processors and different on-chip communication strategies.

The proposed methodology is different from exhaustive-search approaches such as branch-and-bound and ILP. Such methods often allow a more arbitrary cost function, but are not scalable because their runtime and memory requirement can exponentially grow. Simulated annealing and genetic algorithms can help reduce the runtime, but they do not deliver high-quality solutions. Our algorithm is both exact and versatile, while it is pseudopolynomial with respect to problem size.

[1]A cost function is realistic iff out of two solutions with identical workload distributions, the one with smaller interprocessor communication is always favored.

Our second contribution is to extend the exact task assignment algorithm to a strictly polynomial approximate task assignment. It 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.

Third, we develop a heuristic task assignment algorithm for pipeline software synthesis for three or more processors. We recursively use our dual-core algorithm for this purpose.

In order to validate practicality of our theoretical contributions, we integrated the proposed algorithms in MIT StreamIt 2.1 compiler framework [Gordon et al. 2002], and generated executable code for multicore platforms. We emulated several multicore architectures on an FPGA, and executed parallel applications. Subsequently, we measured the throughput of running applications on emulated design. Empirical measurements validated the effectiveness of our approach.

Our exact task assignment method for two processors yields 1.7× throughput improvement over single-core baseline, on average. Out of a total six benchmarks, half of them had the same results with both our exact algorithm and StreamIt 2.1, but for the other three benchmarks, our approach yields 25%, 15% and 14% more throughput. In addition, our approximation method offers a range of runtime-throughput tradeoff points. For example, it runs about 3 times faster and requires about 20 times less memory, while results in throughput degradation of only 1% to 5%. On average, the heuristic method yields 13.3%, 45.8% and 14.3% more throughput than StreamIt 2.1, for 3, 4 and 5 processor platforms respectively.

2. RELATED WORK

A number of recent efforts address the problem of multiprocessor software synthesis for SDF-based streaming applications. Thies et al. present StreamIt, a modeling language and compiler, for developing streaming applications [Thies et al. 2002]. Gordon et al. [2002] and Thies et al. [2003] describe a task assignment algorithm for StreamIt. The algorithm partially explores task parallelism and partitions the tasks among multiple processors. Gordon et al. [2006] extend their work and present a heuristic algorithm for acyclic StreamIt task graphs to exploit task, data and pipeline parallelism. As part of the Ptolemy project, Pino et al. [1995] propose a technique to address the combined problem of task assignment and task scheduling for DSP applications modeled as acyclic SDF graphs.

Stuijk et al. [2007] propose a task assignment method for heterogeneous architectures in which, tasks are first sorted based on their impact on throughput. Subsequently, a greedy method assigns one task at a time to the processor with the least workload. Cong et al. [2007] present an algorithm for assignment of acyclic task graphs onto application specific softcore multiprocessor systems. The method is similar to technology mapping in logic synthesis domain. 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. For pipelined network processors executing a sequential code (e.g., C), a program partitioning algorithm is proposed in Yu et al. [2007]. Heuristic graph bipartitioning algorithm is recursively applied to the task graph in order to balance instruction counts across the processors, then a refinement algorithm tries to balance the computation workload.

Unlike existing techniques, our work contributes to the state of the art by formally delivering the optimal solution. We exploit planarity of target task graphs to deliver provably optimal results with reasonably low complexity. Some SDFs, such as those specified in StreamIt, are inherently planar. In addition, we develop a planarization transformation to handle nonplanar task graphs. Our technique supports many hardware-inspired throughput estimation functions during software synthesis. That is, it is capable of minimizing realistically arbitrary cost functions that model execution period in a specific hardware implementation.

Some researchers have taken the approach of exploring the design space via enumeration. Ma et al. [2005] propose a task interleaving algorithm for acyclic task graphs. First, an exhaustive search algorithm for three processors is developed based on which, the technique is extended heuristically to handle architectures with more number of processors. Resource allocation for acyclic graphs on heterogeneous architectures is studied in Hu and Marculescu [2005]. A branch-and-bound algorithm is presented to minimize communication overheads between tasks. A multiobjective genetic task assignment algorithm for applications modeled as Kahn Process Network to a heterogeneous architectures is presented in Erbas et al. [2006]. It considers power consumption, computation workload and communication overheads; however, it requires about 1,000 generations to converge. Approaches based on exhaustive search of the design space are typically not scalable because the runtime and/or memory requirement grow exponentially with problem size.

Our work also relates to a number of theoretical results on graph partitioning. The reason is that at the core of our proposed task assignment technique is a graph bipartitioning algorithm, which operates on planar graphs. Note that the proposed task assignment is not limited to planar task graphs because nonplanar task graphs are planarized before applying the partitioning algorithm (Section 7.4). Graph bipartitioning remains to be intractable for planar graphs [Karpinski 2002], and thus, it admits no polynomial optimal algorithm unless $P = NP$. Several bipartitioning algorithms exist for planar graphs that run in pseudopolynomial time. Our method offers value in comparison to such methods for two main reasons. Firstly, it exploits specific features of application task graphs to improve time complexity of the general case. Secondly, it supports a broad class of cost functions, while existing techniques focus on finding balanced cut (also known as min cut), or minimum quotient cut (also known as sparse cut).[2]

[2]Balanced cut is a subset of edges whose removal would create two equally sized partitions. Minimum quotient cut minimizes the cost function $\frac{C}{min(W_1, W_2)}$ where, W_1 and W_2 refer to total weight of nodes in each partition, and C denotes the cost of cut edges. From a practical viewpoint, W_i and C model computation workload and communication latency, respectively.

Bui and Peck [1992], a bipartitioning algorithm is presented to find the balanced cut of an unweighted planar graph in $O(b^2N^3 2^{4.5b})$ time, where N is the number of nodes in the graph, and b is the number of cut edges. The authors also present another algorithm that runs in $O(k^2N^3 2^{3k})$ time if each biconnected component of the planar graph is at most k-outerplanar. In Rao [1992], two 1.5-approximation and 3.5-approximation algorithms are presented that find minimum quotient cut in $O(N^2 \times \min(W, C))$ and $O(N^2 \times \min(N, \log WC))$ time, respectively. Rao's work has been adapted and improved by many researchers. In Park and Phillips [1993], an exact algorithm and a 3.5-approximation algorithm is presented for finding minimum quotient cut that run in $O(N^3W)$ and $O(N^2 \times \min(\sqrt{N}, \log W \times C))$ time, respectively.

A 2-approximation algorithm for finding 2/3-balanced cuts is proposed in [Garg et al. 2000], which runs in $O(N^4W^2C \log(NW^2C))$ time. An $O(\log N)$-approximation algorithm for finding minimum bisection in polynomial time is developed by Feige and Krauthgamer [2002]. A closely related problem is that of vertex-cut partitioning (i.e., dividing a graph by removing some of the vertices instead of edges). It is possible to transform variants of edge-cut and vertex-cut partitioning problems to each other [Rao et al. 2003]. Initially, the existence and construction of vertex-cut in planar graphs was studied by Lipton and Tarjan [1979], and based on that, a number of other algorithms for vertex-cut partitioning have been designed [Rao et al. 2003; Aleksandrov et al. 2007].

Our graph bipartitioning algorithm can be viewed as a customized extension to the work of Park and Philips [1993], in that we exploit the practical features of software synthesis problem to reduce time complexity for the restricted class of subject graphs, and to handle a variety of arbitrary partitioning objectives.

3. BACKGROUND AND PRELIMINARIES

3.1 Application Model

Many researchers have investigated appropriate abstractions for modeling of streaming applications that are meant to be implemented as parallel software modules. While the purpose of this article is not delving into model of computation and programming languages research, we briefly discuss several outstanding choices for modeling streaming applications.

It is a widely accepted assertion that coarse grain parallelism-extraction is a very difficult problem, and hence, sequential single-threaded languages, such as standard `C`, are not appropriate choices for modeling concurrent applications. A number of leading experts believe that thread-based application development in general, is not a productive and reliable method of developing concurrent software [Gordon et al. 2002; Lee 2006]. A possible alternative would be to represent the applications in an actor-oriented model, where coarse-grain parallelism is explicit, tasks co exist and communicate with one another using a governing protocol [Zhou et al. 2007; Stuijk and Basten 2008].

We adopt a task graph application model whose variations are widely used in embedded systems community [Thoen and Catthoor 2000; Pimentel et al. 2001; Balarin et al. 2003]. Task graphs conform to the general notion of actor-oriented

computing. In this model, applications are represented with a directed acyclic graph (DAG) whose nodes represent tasks. Edges of the DAG represent inter-task data dependencies. Tasks are atomic, in other words, their corresponding computation is specified in sequential semantics and hence, intra-task parallelism is not exploited.

Our loosely defined notion of task graph is very similar to acyclic Kahn process networks [Kahn 1974] in which, tasks communicate asynchronously using unidirectional FIFO links, and receiving tasks block on empty input links. This model has a dataflow nature, which fits streaming applications very well. Several other dataflow-based models, such as synchronous dataflow (SDF) [Lee and Messerschmitt 1987b] can be represented with task graphs. In case of SDF, for example, a node in the task graph would correspond to its associated node in SDF repeated i times, where i is the number of appearances of that node in SDF static schedule.

Each task is associated with a computation that incurs latency when run on a processor. We use the notion of task *workload* (w) to refer to the time that the associated computation takes on a certain processor (Section 3.2). We use the terms w_v and $w(v)$ interchangeably to refer to workload of task v. Moreover, edges of the task graph are associated with data that needs to be communicated between two adjacent tasks. We use the term *communication* latency (c) to refer to the time it takes for a processor to send or receive the associated data onto the network in a certain architecture (Section 3.2). We use the terms c_e and $c(e)$ interchangeably to refer to communication latency of edge e. For a given application, values of w and c are estimated with respect to given target architecture.

Unlike general SDF, however, our target task graphs contains no cycles. Thus, we collapse cycles in SDF-based specifications into single nodes to represent the application as an acyclic task graph. Such task graphs are an important subset of general SDFs because, as we will see in Section 9, many important applications are represented in this form.

3.2 Abstract View of Hardware

We aim at implementing a given streaming application as software modules running on parallel processors. To depart from the problems associated with shared states among threads and to move closer to a robust actor-oriented implementation [Lee 2006], we envision processors to work with distributed (separate) memory spaces. Therefore, synthesized software processes would need to directly send and receive messages to synchronize.

The streaming nature of the application suggests pipelined execution as a favorable design choice for throughput maximization. For pipelined software execution, the hardware *seems* to be composed of processors connected using unidirectional FIFO channels (links). For the optimal task assignment algorithm, there is only two processors in the system. Later, in Section 7.3, we will extend our discussion to arbitrary number of processors and develop a heuristic task assignment.

Figure 1(a) illustrates the hardware model with two processors and a FIFO link. Note that this is the abstract view of the hardware, and not necessarily the available physical hardware. For example, a shared-memory multicore

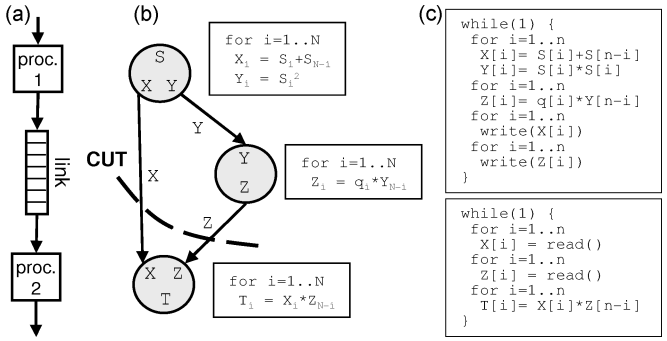


Fig. 1. (a) Abstract dual-core architecture. (b) Example task graph. (c) Generated software for depicted task assignment.

architecture can realize the abstract view by implementing interprocessor link in shared memory. In such architectures, unidirectional FIFO channels are implemented as arrays in the shared memory space. The access to the array has to be synchronized via a conventional locking mechanism. Another example implementation would be a parallel platform with on-chip static routers. We abstract out the implementation of message passing and discuss the work with respect of our abstract view of hardware (Figure 1(a)). We temporarily use this abstracted communication model, and later in Section 7.1, we will discuss how our proposed algorithm is capable of considering implementation details.

We temporarily assume that the processors are identical in that they have the same clock speed and the same instruction set architecture, or essentially the same computation power. Later, in Section 7.2, we discuss extensions to our technique to handle heterogeneous processors. A number of existing parallel platforms utilize distributed memory spaces and on-chip communication network for data transfer. Examples include AsAP [Yu et al. 2006] and RAW [Taylor et al. 2004].

3.3 Software Synthesis

Given a set of tasks allocated to processor p, we synthesize the code running on p by combining computations of tasks according to the given schedule, followed by adding communication interfaces (i.e., architecture `read` and `write` communication primitives).

Figure 1 shows an example, in which the top two nodes (tasks) are assigned to the first processor and the bottom node to the second processor. Let us examine the generated code for processor 1. We first add a `read` function to read the input data and initialize S. Then, we insert tasks' internal code to perform the computation of node `a` followed by the node `b`. Finally, `write` functions are inserted in the code to output the results X and Z to the second processor.

Notice that we implicitly assume that some schedule is present to order the tasks assigned to the same processor. The schedule ensure that tasks are combined considering their dependencies. For example, node `b` has to come only after node `a` because it has data dependency to node `a`. Note that our model is based on streaming applications that are scheduled statically.

3.4 Performance Model

We define the performance to be the steady state throughput of synthesized application. Performance of our abstract hardware is affected by a number of factors such as tasks assigned to each processor, tasks schedule, and channel buffer depth. We assume that interprocessor channel buffer is deep enough to not play a role in steady-state performance [Stuijk et al. 2006].

For our target applications, task firing schedule can be determined statically for the entire task graph before task assignment. Once task assignment is performed, a subset of tasks are assigned to a given processor. We assume that the original schedule determines the ordering of tasks assigned to the processor. Hence, a new schedule for tasks assigned to a processor is not needed.

The performance of synthesized pipeline code is determined by the slowest part in the pipeline. It follows that the execution period[3] is equal to $Max(W_1 + C, C + W_2)$, where W_1 and W_2 refer to total computation workload assigned to processor 1 and 2, and C denotes interprocessor communication latency. Note that in the synthesized software, interprocessor data communication is aggregated in time to better highlight the impact of each term on performance.

Specifics of the platform selected to implement the abstract hardware play a key role in estimation of both the three aforementioned terms (W_1, W_2, C) and their impact on performance. For simplicity, we develop our technique assuming that $Max(W_1 + C, C + W_2)$ determines application performance. In Section 7.1, we generalize our task assignment technique to handle other implementation-specific cost functions.

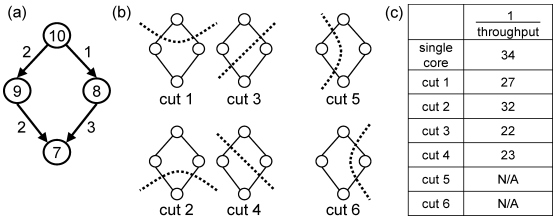
4. TASK ASSIGNMENT PROBLEM

In order to compile a task graph $G(V, E)$ onto our abstract architecture, we need to assign every node (task) in G to one of the processors. In other words, every node $v \in V$ needs a processor to execute the computation represented by v. This procedure is called *task assignment*. Since the number of tasks is usually larger than the number of processors, each processor has to execute a group of tasks. Given a task graph G, the problem is to assign a subset of tasks to each processor to maximize application throughput.

4.1 Motivating Example

Consider the example task graph in Figure 2(a), in which tasks and edges are annotated with their corresponding workload and communication latency, respectively. The application is to be mapped onto the target dual-processor architecture. In this example, there exist a total of six possible cuts in the graph ($C_1, \ldots, C_6$), which are depicted in Figure 2(b). The goal is to pick the cut which maximizes throughput (i.e., minimizes execution period).

[3]Execution period is equal to one over throughput. It is the time period between successive arrivals of output data samples.



	$\frac{1}{\text{throughput}}$
single core	34
cut 1	27
cut 2	32
cut 3	22
cut 4	23
cut 5	N/A
cut 6	N/A

Fig. 2. (a) An example task graph. Vertex and edge annotations represent task workload and communication latency. (b) Possible cuts for task assignment. Nonconvex cuts 5 and 6 are not valid choices. (c) Resulting execution period.

If we partition the task graph with cut C_1, the top node is assigned to processor one and the other three nodes to processor two. One full cycle of the entire application takes: 10 unit time for computation carried out in the first processor plus $1 + 2 = 3$ unit time to `write` data onto the link, and $9 + 8 + 7 = 24$ unit time for computation carried out in the second processor plus 3 unit time to `read` the same data. Since this particular cut partitions the graph into a pipeline-like structure (starting from the first processor and ending in the second processor), we can issue one new task every $max(10+3, 3+24) = 27$ units time. In other words, the execution period is 27 and the application throughput is $\frac{1}{27}$.

Figure 2(c) shows the same analysis for all possible cuts. C_1 and C_2 are not taking that much advantage of the parallel architecture as most of the load is on one processor. Both C_3 and C_4 have a high throughput, since they both divide the computation workload almost even. However, despite the fact that C_4 has a more balanced workload than C_3, its throughput is lower. This is because the higher communication latency of C_4 negatively effects the overall throughput ($max(18 + 5, 5 + 16) = 23$).

C_5 and C_6 are called nonconvex cuts and are not considered as possible candidates. Nonconvex cuts create a cyclic dependency among processors, which greatly complicate task scheduling and also may reduce throughput [Atasu et al. 2003; Cong et al. 2007; Yu et al. 2007]. For example, in case of C_6, the synthesized code for processor one should have a `read` at the beginning to provide input data for the top node, and after some computations another `read` in the middle of the code for the bottom node. In such cases, finding an optimal ordering of tasks and `read`/`write` operations for efficient pipelined execution becomes very difficult. In addition, it requires a bidirectional link.

As highlighted in the above example, task assignment has a significant impact on the overall application throughput. Both computation workloads and communication latencies need to be considered during task assignment. The example also points out that nonconvex cuts should be avoided to guarantee existence of a feasible pipelined schedule.

5. TASK ASSIGNMENT VIA GRAPH PARTITIONING

In this section, we formulate throughput-driven task assignment problem as graph partitioning. Then, we proceed to develop our task assignment algorithm,

which considers computation workloads assigned to processors and interprocessor communication traffic. We refer to our algorithm as TAP for "throughput-driven application partitioning."

Note that the discussion of this section is restricted to parallel platforms with two processors. As a result, the corresponding graph partitioning problem becomes a bipartitioning instance. Later, in Section 7.3, we present a heuristic method to handle arbitrary number of partitions in the graph, or equivalently, arbitrary number of processors. In addition, we restrict our discussion in this section to partitioning planar task graphs. In Section 7.4, we extend our technique to address generic task graphs.

From a theoretical point of view, TAP is an exact algorithm (i.e., provably optimal) for maximizing bipartition throughput. Accurate dependency of throughput to either workload distribution and/or interprocessor communication traffic depends on specifics of target platform and software synthesis. Temporarily, for the sake of discussion, we assume that application throughput is determined by the slowest of (a) computation on core 1 plus the time needed to write data into the channel, and (b) computation on core 2 plus the time needed to read data sent by core 1 from the channel. In other words, task assignment cost function is the maximum of these two terms. In Section 7.1, we look beyond this specific cost function and discuss extensions to TAP to handle other platform-inspired cost functions.

5.1 Formalization and Definitions

Graph G is *planar* if and only if it can be drawn on a two-dimensional (2D) plane in such a way that no two edges cross. A crossing-free drawing of a planar graph G is also referred to as an *embedding* of G in 2D plane. A given embedding of a planar graph defines *faces*, that are regions on 2D plane that border edges of G.

Similar to a planar graph, a *planar DAG* $G(s, t)$, is a directed acyclic graph that has crossing-free drawings on 2D plane, where s is the source vertex and t is the sink vertex. If there are multiple sources (sinks), we add a dummy source (sink) connected to all sources (sinks) to make G a single-source (single-sink) DAG. Figure 3(a) shows an example planar DAG embedded in 2D plane.

A given application is viewed as a planar directed acyclic task graph $G(V, E)$, where V and E denote the set of vertices and edges of the graph, respectively. Each vertex $v \in V$ is associated with the workload of the corresponding task $w(v)$. Additionally, every edge $e \in E$ is annotated with the corresponding inter-task communication latency $c(e)$. Note that $c(e)$ denotes the communication latency when the sending and receiving tasks are assigned to different processors. We use the notions of e_{ij}, $e(i, j)$ and (i, j) interchangeably to refer to the edge that connects vertex i to vertex j in the graph.

Cut C in G is defined as a subset of edges in G that divide the graph into two smaller connected subgraphs, G_1 and G_2. $W_G = \sum_{v \in V} w(v)$ is total workload of application tasks, which determines application throughput in a single-processor implementation. Similarly, W_{G_1} and W_{G_2} denote the total workload of tasks in subgraph G_1 and G_2, respectively. The total communication latency of

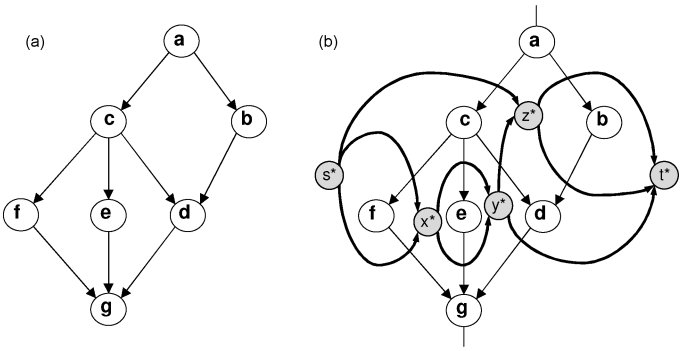


Fig. 3. (a) Example planar DAG $G(a, g)$. (b) its dual graph $G^*(s^*, t^*)$.

cut edges C is represented with $C_C = \sum_{e \in C} c(e)$, which represents total latency of transferring (both writing and reading) data between the processors.

5.2 Problem Formulation

The task assignment problem is to allocate processing resources to constituting tasks of the application, and subsequently, generate software such that the application throughput is maximized. Each partitioning of the task graph implies a task assignment for software synthesis. Hence, in the graph partitioning domain, the problem transforms to finding the best convex cut in the application task graph. The convexity constraint ensures unidirectional dependency among processors, which simplifies pipelined scheduling of tasks and improves throughput [Atasu et al. 2003; Cong et al. 2007].

Specifically, we seek to maximize the overall throughput by finding a convex cut C in graph G that minimizes the cost function $Q_C = max(W_{G_1} + C_C, C_C + W_{G_2})$. This cost function, inspired by our discussion in Section 3.4, jointly considers the impact of workload imbalance and inter-partition communication on throughput. In Section 7.1, we discuss extensions to other cost functions.

5.3 Transformations and Properties

Given a planar graph $G(V, E)$, its *dual* graph $G^*(V^*, E^*)$ is well defined and is constructed as follows: For each face of G, including the external face, a vertex is introduced in V^*. For each edge $e \in E$, a new edge is introduced in E^* to connect the two dual vertices that correspond to two faces of G meeting at e. Note that G^* is also a planar graph. Edges in E are in one-to-one relation with edges in E^*, faces of G relate to vertices of V^*, and vertices of V relate to faces of G^*.

Similarly, we construct dual of a planar DAG $G(s, t)$ and denote it by $G^*(s^*, t^*)$, where s^* and t^* are G^* source and destination vertices (Figure 3(b)) [Angelini et al. 2007]. The trick here is that for the purpose of dual graph construction, we assume there are two infinite edges, one from infinity to s, and one from t to infinity. Without the two infinite edges, there would be only one

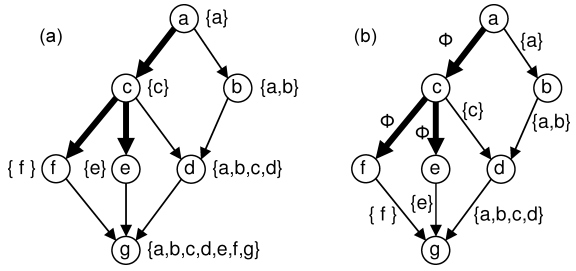


Fig. 4. Example planar DAG $G(a, g)$. Λ-edges are marked bold. (a) $S(v)$ set for vertices of G. (b) $S(e)$ set for edges of G.

external face, but now there are two; one on the left[4] and one on the right side of the planar DAG $G(s, t)$. Therefore, we have two vertices for our two external faces, s^* is defined as the one on the left, and t^* on the right. Note that edges of G^* are all directed from s^* to t^* and there is no cycle in G^*.

Any convex cut C in the planar DAG $G(s, t)$ corresponds to a simple path P from s^* to t^* in G^*. Path P is simple in that it does not revisit a vertex $v \in V^*$. Based on the duality, P does not visit a face $f \in G$ more than once. Intuitively speaking, s and t belong to different partitions, and any path from s to t in G is cut exactly once. Figure 5(a) shows a valid cut in G and Figure 5(b) shows its corresponding simple path in the dual graph G^*.

Definition 5.1. In planar DAG $G(s, t)$ with a fixed embedding, we call edge e_{uv} a Λ-*edge* iff e_{uv} is not the rightmost outgoing edge of u. Note that with a particular embedding in mind, the *rightmost* outgoing edge is well defined for node u. $E_\Lambda \subset E$ is the set of all Λ-edges in G. Figure 4(a) shows a planar DAG G with its Λ-edges marked bold.

Definition 5.2. For vertices u and v in a planar DAG $G(s, t)$, u *reaches* v iff there is a directed path P with no Λ-edges from u to v. We use the notions $u \rhd v$ to denote that u reaches v. Their connecting path is referred to as $P_{u\to v}$ that by definition does not include any Λ-edges. Every vertex reaches itself. The set of all vertices that reach a vertex v is called $S(v)$. Formally:

$$u \rhd v \iff \exists P_{u\to v} \forall e \in P : e \notin E_\Lambda$$
$$S(v) = \{u_i \in V \,|\, u_i \rhd v\}$$

Figure 4(a) shows $S(v)$ sets for all vertices of the graph in which a reaches b, d, and g, but not c, e, and f.

COROLLARY 5.3. *If $u \rhd v$ and $v \rhd w$ then $u \rhd w$.*

COROLLARY 5.4. *All vertices of G reach the destination vertex, i.e., $S(t) = V$.*

PROOF: because $G(s, t)$ is a connected graph. □

[4]Since the graph is planar, *left* and *right* is well defined because we fix an embedding on the 2D plane. Note that it is not required to actually calculate X and Y coordinates because a simple ordering of incoming and outgoing edges of every vertex is enough for our purpose.

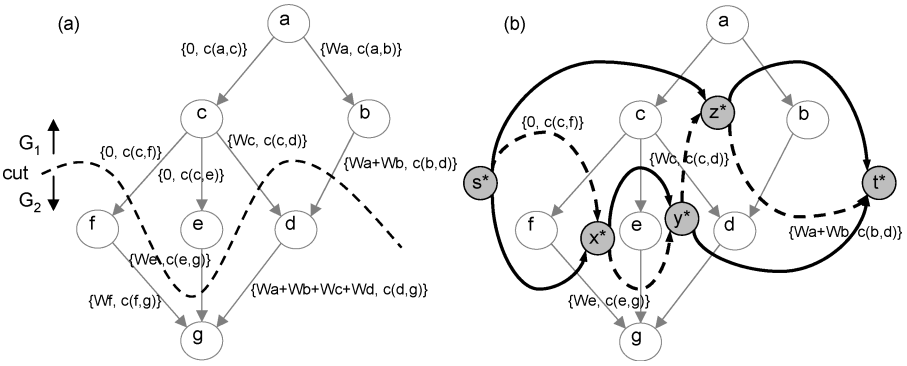


Fig. 5. (a) Graph G and its cut properties. (b) Paths from s^* to t^* in G^* correspond to convex cuts in G.

Definition 5.5. $S(e_{uv})$ is the set of all vertices that reach edge e_{uv}, and is defined to be:

$$S(e_{uv}) = \emptyset \text{ if } e \in E_\Lambda$$
$$S(e_{uv}) = S(u) \text{ otherwise.}$$

Figure 4(b) shows $S(e)$ sets for all the edges in graph G. For example, $S(e_{bd})$ is equal to $S(b)$.

LEMMA 5.6. *For every vertex $v \in V$ with incoming edges $e(u_1, v), \ldots, e(u_n, v)$, the following properties hold:*

$$I) \bigcup_{i=1}^{n} S(e(u_i, v)) = S(v) - \{v\}$$
$$II) \forall 1 \le i, j \le n, i \ne j : S(e(u_i, v)) \cap S(e(u_j, v)) = \emptyset$$

That is, the union of $S(e)$ sets of incoming edges of a vertex is equal to the $S(v)$ set of that vertex excluding itself; and all distinct $S(e)$ sets have empty intersections. As an example, consider vertex d in Figure 4, where $S(d) = \{a, b, c, d\}$, $S(e_{cd}) = \{c\}$ and $S(e_{bd}) = \{a, b\}$.

PROOF: (I) because for a vertex v, all u_i vertices either reach v or have empty $S(e(u_i, v))$. The term $\{v\}$ is needed because v reaches itself. (II) because based on the definition of $S(e_{uv})$, only one of the outgoing edges of u have the value of $S(u)$ and the rest receive zero. □

We transform the application task graph by annotating its edges with additional information. Specifically, we annotate edges with workloads of selected vertices. Each node $v \in V$ propagates the sum of its workload $w(v)$ and the propagated workload from its incoming edges to its rightmost edge. Figure 5(a) shows an example of the transformation. Every edge $e \in E$ is annotated with $w(e)$ that is formally defined as:

$$w(e) = \sum_{v \in S(e)} w(v)$$

After the transformation, there would be two values associated with each edge: workload and communication latency. For instance in Figure 5(a), edge e_{ac} has both communication latency $c = c(e_{ac})$ and workload $w = 0$

annotations. This transformation simplifies evaluation of cuts by decoupling their cost (according to cost function) from graph structure. That is, one only needs to consider a cut, including the values annotated on edges in the cut, to evaluate its cost. Note that before the transformation, information about graph structure was needed to evaluate a given cut.

We define workload of a cut W_C, to be sum total of workloads of the edges in cut: $W_C = \sum_{e \in C} w(e)$. Also, recall that workload of a subgraph G_1 was defined as $W_{G_1} = \sum_{v \in G_1} w(v)$. Formally:

THEOREM 5.7. *For every convex cut C in G,* $W_{G_1} = W_C$.

PROOF: We make a small change to G_1 to make it a single-sink planar DAG. Assume that there is a dummy sink vertex t' added below the cut in G_1, and all cut edges are connected to this dummy vertex. Now, the modified G_1 becomes a planar single-source and single-sink DAG $G_1(s, t')$. Graph $G_1(s, t')$ has exactly the same $S(e)$ and $S(v)$ values as the corresponding part of original graph G, because $S(e)$ and $S(v)$ values depend only on graph structure between source node and the cut. Hence, the inserted dummy sink does not affect $S(e)$ and $S(v)$ values in G_1. Thus, according to corollary 5.4:

$$V(G_1) = S(t')$$

According to Lemma 5.6, $S(t')$ is equal to union of $S(e)$ sets of all t' incoming edges. Note that incoming edges of t' are the cut edges. Therefore, $S(t') = \bigcup_{e \in C} S(e)$ which leads to $V(G_1) = \bigcup_{e \in C} S(e)$. Based on the second part of Lemma 5.6, $S(e)$ sets are nonoverlapping. Therefore, we can safely substitute each vertex with its workload value in the above equation, and arrive at another equality:

$$\sum_{v \in G_1} w(v) = \sum_{e \in C} \sum_{v \in S(e)} w(v)$$

And based on the definition of edge workloads $w(e)$, we have:

$$W_{G_1} = \sum_{v \in G_1} w(v) = \sum_{e \in C} w(e) = W_C \quad \square$$

Figure 5(a) illustrates an example. Sum of the workloads of cut edges (edges that cross dotted lines) is $W_C = (0) + (W_e) + (W_c) + (W_a + W_b)$, which is equal to sum of the workloads of all vertices in the upper partition $W_{G_1} = W_a + W_b + W_c + W_e$.

Theorem 5.7 implies that we can obtain both the computation workload of the upper-partition of a cut C, and the sum total of inter-partition communication latency, by examining cut C alone. Subsequently, the cost function $Q_C = max(W_{G_1} + C_C, C_C + W_G - W_{G_1})$ can be written as $Q_C = max(W_C + C_C, C_C + W_G - W_C)$. Since W_G is constant for a given application and graph transformation (edge annotations) is performed only once, evaluation of different cuts does not need successive analysis of the entire graph. Cuts can be evaluated only by taking into account the information annotated on cut edges.

5.4 Throughput-Driven Application Partitioning Algorithm

We proceed to develop our optimal throughput-driven application partitioning (TAP) algorithm in this section. We will utilize the definitions and properties

discusses in previous subsections to formally establish algorithm's correctness and optimality.

We construct the dual graph $G^* = (V^*, E^*)$ from the original task graph $G = (V, E)$. Assuming that edge $e \in E$ corresponds to edge $e^* \in E^*$, edge e^* is annotated with workload and communication latency of edge e. That is:

$$c(e^*) = c(e) \text{ and } w(e^*) = w(e)$$

Based on duality, a convex cut C in G corresponds to a simple path P in G^* from s^* to t^* (Figure 6(b)). Therefore:

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

where C_P and W_P are equivalent to C_C and W_C but are calculated from path P that corresponds to cut C. More formally, $W_C = W_P = \sum_{e^* \in P} w(e^*)$ and $C_C = C_P = \sum_{e^* \in P} c(e^*)$.

The number of convex cuts in G, or equivalently the number of simple paths in G^*, can grow exponentially with respect to graph complexity. In order to tractably find the best cut, we expand dual graph G^* to graph $G^\dagger = (V^\dagger, E^\dagger)$, which is formally constructed as follows:

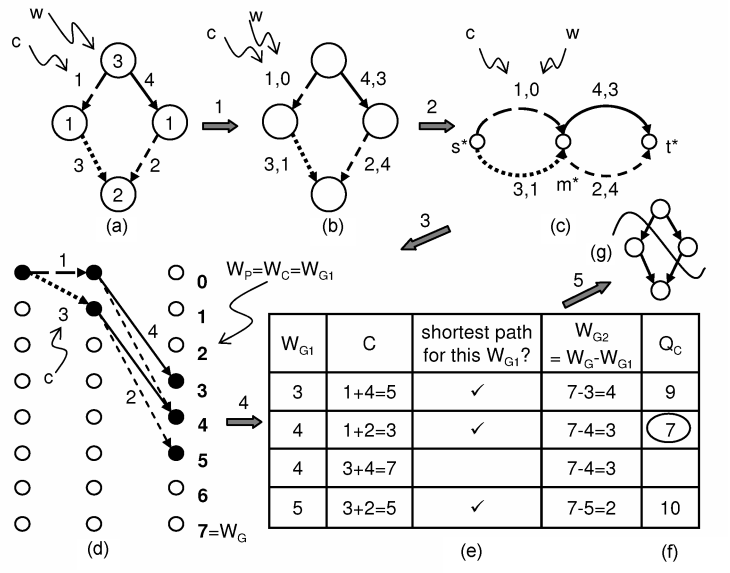
$$V^\dagger = \{v_w^* : v^* \in V^*, 0 \leq w \leq W_G\}$$
$$E^\dagger = \left\{\left(u_w^*, v_{w+w(u^*,v^*)}^*\right) : (u^*, v^*) \in E^*\right\}$$

That is, $G^\dagger$ is constructed by duplicating G^* vertices W_G times. Vertex u_p^* is connected to vertex v_q^* in $G^\dagger$ if and only if u^* and v^* are connected with an edge having $q - p$ units of workload in G^*. Note that for a given edge $e^* \in E^*$ connecting u to v in G^*, and a given constant w_0, there is a unique edge in $E^\dagger$ that connects $u_{w_0}^*$ to $v_{w_0+w(e^*)}^*$ in $G^\dagger$. Edges of $G^\dagger$ have no direct workload annotations because their *placement* contains the workload information. They are, however, annotated with the communication latency of the corresponding edge in G^*.

It is interesting to observe that any path from s^* to t^* in G^* corresponds to one and only one path from s_0^* to t_ω^* in $G^\dagger$, where t_ω^* is one of the nodes corresponding to t^* in $G^\dagger$. Moreover, such path has total workload of ω. Intuitively, the subscript difference between u_p^* and v_q^* denotes increase of $q - p$ units in path workload, should their connecting edge be taken in path in $G^\dagger$. Since we start at s_0^* and arrive at t_ω^*, the total workload of path edges (W_P) is ω.

Figure 6 illustrates the situation in which, an example task graph G is transformed to graph G^*. Subsequently, graph $G^\dagger$ for the example task graph is constructed. Note that there is a one to one correspondence between convex cuts in G, source to destination paths in G^*, and paths in $G^\dagger$.

Recall that the objective is to find a path P in G^* from s^* to t^* that minimizes Q_P, where $Q_P = max(W_P + C_P, C_P + W_G - W_P)$. Let us assume path P in G^* corresponds to path $P^\dagger$ in $G^\dagger$. Since the index of the vertex where $P^\dagger$ ends in $G^\dagger$ is equal to W_P, we do not need to calculate this value. In addition, since communication latency of an edge in $G^\dagger$ is equal to communication latency of the corresponding edge in G^*, $C_P = C_{P^\dagger}$.



W_{G1}	C	shortest path for this W_{G1}?	W_{G2} = W_G-W_{G1}	Q_C
3	1+4=5	✓	7-3=4	9
4	1+2=3	✓	7-4=3	7
4	3+4=7		7-4=3	
5	3+2=5	✓	7-5=2	10

Fig. 6. (a) planar DAG G. (b) G after moving the workload values to edges. (c) dual graph G^*. (d) expanded graph $G^\dagger$. (e) Shortest path cuts for each W_{G_1} value in $G^\dagger$ and also the minimum Q_C among all shortest paths. (f) The cut with minimum cost is selected.

Let us treat edge communication latencies as edge distances in $G^\dagger$. It follows that for a given workload target ω, the shortest path from s_0^* to t_ω^* in $G^\dagger$ minimizes $C_{P^\dagger}$. The corresponding path in G^* has the minimum C_P value and hence, optimizes the cost function for the given workload target. In other words, such path gives the best partitioning such that the top partition has workload of ω.

Since we do not know the proper choice of ω at the beginning, we run the single-source shortest path $G^\dagger$ from s_0^* to all possible destinations $t_\omega^*(\forall 0 \leq \omega \leq W_G)$. The shortest path prunes many possible cuts that will not result in minimized cost function, and hence, we are left with only W_G paths. Each shortest s_0^* to t_ω^* path gives the best possible cut with ω units of workload in the top partition. Subsequently, we calculate the minimum possible value of $Q_P = max(W_P + C_P, C_P + W_G - W_P)$ at each of t_ω^* nodes, and select the destination node that globally minimizes Q_P (Figure 6).

5.5 Complexity

Task graphs and subsequent graphs constructed from task graphs are directed acyclic graphs. Hence, the complexity of discussed transformations and the shortest path algorithm linearly depend on the number of edges in the subject graphs. Note that on directed acyclic graphs, single source shortest path can be implemented using topological sort and thus has linear time complexity [Cormen et al. 2001].

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, $G^\dagger$. There are at most $N \times W_G$ nodes

in $G^{\dagger}$, where N is the number of nodes in the application task graph and W_G is the total workload of application tasks. Thus, our algorithm has the time complexity of $O(N.W_G)$, which is considered pseudopolynomial. The following theorem proves that the problem is NP-Complete, and therefore, algorithms with strictly-polynomial complexity do not exist unless $P = NP$.

THEOREM 5.8. *The decision problem corresponding to the graph partitioning problem formulated in Section 5.2 is NP-Complete.*

PROOF: Membership in NP is obvious because any given solution can be verified in polynomial time. We reduce an instance of set partition NP-Complete problem [Garey and Johnson 1990] to an instance of our graph partitioning problem. The set partition problem asks the following question: can a given set of arbitrary integer numbers be partitioned into two disjoint subsets such that the summation of numbers in partitions are equal?

For a given set partition instance, we create the corresponding graph partitioning instance in the following manner: For each number in the set, we insert a task in the graph with the same workload. Tasks are not dependent and can run in parallel. We connect all tasks to a dummy source (sink) with edges that have zero communication latency. Maximizing throughput for such a task graph would answer the set partitioning question. Therefore, our task assignment problem is NP-Complete. □

In practice, task workload values can be normalized so that W_G value remains relatively small. Therefore, pseudopolynomial time complexity does not impose a real constraint on practicality of our approach. Pseudopolynomial time algorithms display polynomial time behavior unless we have to deal with exponentially large numbers which are uncommon in many practical settings [Garey and Johnson 1990].

6. APPROXIMATE TASK ASSIGNMENT

In this section, we present an approximation method for task assignment with strictly linear 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 $G^{\dagger}$ in the exact algorithm. The number of vertices in the expanded graph $G^{\dagger}$ is reduced to $O(N \log W_G)$ from the original $O(N.W_G)$, 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 6.1. 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 $G^{\dagger}$. Thus, for a path P from s^* to t^* in G^*, the approximation function is applied k times, where k is the number of edges in path P. Figure 7 shows an example in which, path P is marked with dashed lines. All paths start from s_0^*.

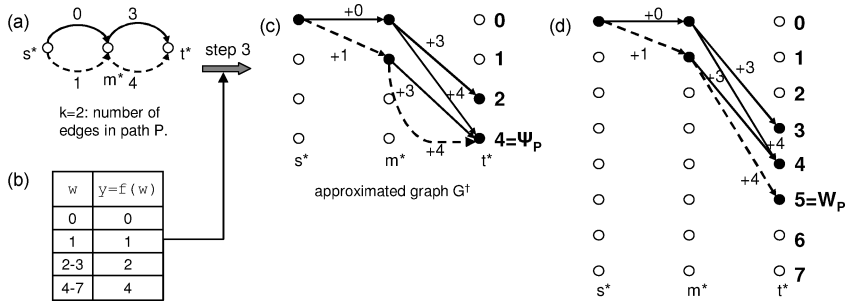


Fig. 7. Example: (a) Dual graph G^* from Figure 6, (b) An illustrative approximation function with $1+\delta=2$. (c) The resulted graph $G^\dagger$ with approximation and (d) without approximation.

In path P, workload of the edge from s^* to m^* is equal to 1, and based on Figure 7(b), $f(0+1)=1$; therefore, the first edge of this path is from s_0^* to m_1^*. Now, the next edge starts from m_1^*. $1+4=5$ and $f(5)=4$; therefore, the second edge is from m_1^* to t_4^* (Figure 7(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 7.d). We denote the approximated W_P with Ψ_P.

Therefore, the number of vertices in the graph $G^\dagger$ 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: $0, 1, 1+\delta, (1+\delta)^2, \ldots, (1+\delta)^{\lfloor \log_{1+\delta}^{W_G} \rfloor}$.

LEMMA 6.2. *For approximation function, $y=f(w)$ described in Definition 6.1*

$$\frac{w}{1+\delta} < y \le w$$

PROOF:

$$\lfloor \log_{1+\delta}^{w} \rfloor \le \log_{1+\delta}^{w} < \lfloor \log_{1+\delta}^{w} \rfloor + 1$$
$$(1+\delta)^{\lfloor \log_{1+\delta}^{w} \rfloor} \le (1+\delta)^{\log_{1+\delta}^{w}} < (1+\delta)^{\lfloor \log_{1+\delta}^{w} \rfloor + 1}$$

And thus based on definition of the approximation function $y=f(w)$:

$$y \le w < (1+\delta)y$$

Therefore, we have $\frac{w}{1+\delta} < y \le w$. □

THEOREM 6.3. *For every path P from s^* to t^* in the expanded graph $G^\dagger$, the approximated workload Ψ_P is within the following range from its original value W_P.*

$$\frac{W_P}{(1+\delta)^k} < \Psi_P \le W_P$$

PROOF: Let P_i denote a partial path consisting of the first i edges of path P. For example, in Figure 7(c), P_1 has only one edge (s_0^*,m_1^*), and P_2 has two edges (s_0^*,m_1^*) and (m_1^*,t_4^*). Since k is the number of edges in P, P_k is equal to P. Ψ_{P_i} is approximated workload of P_i and W_{P_i} is its original value. In our example, $\Psi_{P_1}=1$, $\Psi_{P_2}=4$, $W_{P_1}=1$ and $W_{P_2}=5$. Also, let w_i denote workload of the ith

edge in P, e.g., $w_1 = 1$ and $w_2 = 4$. We have:

$$W_{P_1} = w_1, W_{P_i} = W_{P_{i-1}} + w_i, W_P = W_{P_k}$$
$$\Psi_{P_1} = f(w_1), \Psi_{P_i} = f(\Psi_{P_{i-1}} + w_i), \Psi_P = \Psi_{P_k}$$

Now, we use induction. For $k = 1$, $\Psi_{P_1} = f(w_1)$ and $W_{P_1} = w_1$; therefore, based on lemma 6.2

$$\frac{W_{P_1}}{1+\delta} < \Psi_{P_1} \leq W_{P_1}$$

Now, we assume for $k = i-1$ the theorem holds, and we prove it holds for $k = i$:

$$\frac{W_{P_{i-1}}}{(1+\delta)^{i-1}} < \Psi_{P_{i-1}} \leq W_{P_{i-1}}$$
$$\frac{W_{P_{i-1}}}{(1+\delta)^{i-1}} + w_i < \Psi_{P_{i-1}} + w_i \leq W_{P_{i-1}} + w_i$$
$$\frac{W_{P_{i-1}} + (1+\delta)^{i-1} w_i}{(1+\delta)^{i-1}} < \Psi_{P_{i-1}} + w_i \leq W_{P_{i-1}} + w_i$$

Since $1 < (1+\delta)^{i-1}$

$$\frac{W_{P_{i-1}} + w_i}{(1+\delta)^{i-1}} < \Psi_{P_{i-1}} + w_i \leq W_{P_{i-1}} + w_i$$

$W_{P_i} = W_{P_{i-1}} + w_i$; therefore,

$$\frac{W_{P_i}}{(1+\delta)^{i-1}} < \Psi_{P_{i-1}} + w_i \leq W_{P_i} \quad (I)$$

$\Psi_{P_i} = f(\Psi_{P_{i-1}} + w_i)$, based on lemma 6.2

$$\frac{\Psi_{P_{i-1}} + w_i}{1+\delta} < \Psi_{P_i} \leq \Psi_{P_{i-1}} + w_i \quad (II)$$

From (I) and (II), we have

$$\frac{W_{P_i}}{(1+\delta)^i} < \Psi_{P_i} \leq W_{P_i}$$

Therefore, induction is complete and $\frac{W_{P_k}}{(1+\delta)^k} < \Psi_{P_k} \leq W_{P_k}$. Since $P_k = P$, we have $\frac{W_P}{(1+\delta)^k} < \Psi_P \leq W_P$. □

COROLLARY 6.4. *If we set $\delta = \frac{\epsilon}{2F}$ where $0 < \epsilon < 1$ and F is the number of faces in task graph G, then we have $\frac{W_P}{1+\epsilon} < \Psi_P \leq W_P$.*

Because $k < F$ and therefore $(1+\delta)^k < 1+\epsilon$, this means that for every path P from s^* to t^* in the expanded graph $G^\dagger$, the approximated workload Ψ_P is within the above range from its original value W_P.

THEOREM 6.5. *Let $\Omega_P = max\left(\Psi_P + C_P, C_P + f(f(W_G) - \Psi_P)\right)$ denote approximated value of our cost function $Q_P = max(W_P + C_P, C_P + W_G - W_P)$. We have*

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

PROOF: 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 6.4 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 $G^\dagger$ is not more than a factor away from the optimum solution which we can find in the original graph $G^\dagger$.

7. PRACTICAL EXTENSIONS

In this section, we show that our task assignment technique can readily optimize any function of W_{G_1} and C_C that is nondescending in C_C. In addition, we discuss extensions to our basic technique to handle heterogeneous processors, multiple pipeline processors, and nonplanar task graphs.

7.1 Platform-Inspired Cost Functions

In Section 5, we presented our method with cost function $Q_C = max(W_{G_1} + C_C, C_C + W_{G_2})$. However, specifics of target platform might demand other estimations of throughput. For example, in most network-based interconnect architectures, communication latency does not grow linearly with the volume of traffic. In such cases, a more complex function of communication latency must be employed.

Recall that in our algorithm discussed in Section 5.4, the choice of cost function did not play any role in graph transformations and constructions of G^* and $G^\dagger$. Many graph properties hold regardless of the choice of cost function. For example, regardless of the cost function, any s_0^* to t_ω^* path in graph $G^\dagger$ gives a cut in G that has ω units of workload in the top partition.

The choice of cost function only plays a role when it is being evaluated for all $t_\omega^*(0 \leq \omega \leq W_G)$ nodes in $G^\dagger$. Therefore, Q_C can be safely replaced with (almost) any other function of W_{G_1} and C_C. The only constraint is that the selected function must be ascending in C_C, since the underlying assumption is that the shortest s_0^* to t_ω^* path is better than any other s_0^* to t_ω^* path in $G^\dagger$. This is not a restrictive constraint in practice, because out of solutions with identical workload distributions, the one with smaller interprocessor communication is always preferred.

7.2 Heterogeneous Processors

If processing resources are not homogeneous, they can be examined to establish a relative processing performance. For example, if processors share the same architecture but operate at different clock frequencies, their relative processing performance is proportional to their clock frequencies. If processors utilize different architectures, an estimation of relative performance can be carried out to establish a similar performance ratio.

Once relative processing performance is established, its impact on throughput can be readily incorporated into graph partitioning cost function. For

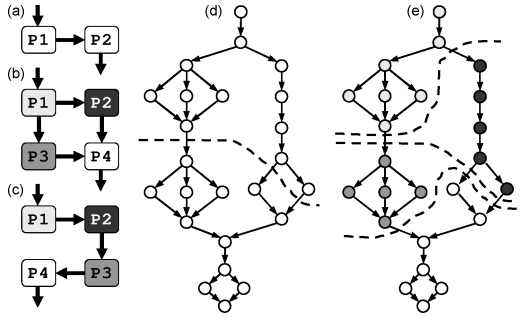


Fig. 8. (a) Pipeline dual-core. (b) Two-dimensional pipeline quad-core. (c) One-dimensional pipeline quad-core. (d) Sample task assignment by exact bipartitioning algorithm. (e) Sample task assignment based on (d) for architectures in (b) and (c).

example if processors utilize the same architecture but first processor runs at half the frequency of the second processor, pipeline period can be estimated as follows:

$$Q_C = max(2 \times W_{G_1}, C_C, W_{G_2}) = max(2 \times W_{G_1}, C_C, W_G - W_{G_1})$$

According to our discussion in Section 7.1, our technique can readily optimize such a cost function during partitioning. Other source of heterogeneity can be similarly incorporated in the cost function. Although, relative performance estimation is harder for some heterogeneity sources such as instruction set differences.

7.3 K-Way Partitioning

We develop a heuristic based on the exact bi-partitioning method of Section 5, to partition the application task graph into arbitrary number (K) of subgraphs. The basic idea is to successively apply our bi-partitioning algorithm $K-1$ times to find $K-1$ cuts that partition the graph into K rather balanced pieces. Partitions of graph G with total workload of W_G each have an ideal workload of $\frac{W_G}{K}$.

Figure 8 is an example on $K = 4$ processors (note that K is not restricted to power of 2). In the example, our exact bi-partitioning algorithm is first used to partition the task graph in two subgraphs (Figure 8(d)), then each of them are further divided in two smaller subgraphs, one for each processor (Figure 8(e)). As shown in Figure 8(b) and 8(c), by recursively applying the exact bipartitioning algorithm, we are able to assign tasks to both one- and two-dimensional pipeline platforms.

To apply the successive bipartitioning, we calculate K_1 and K_2, which denote the number of processors available for tasks of each partition and hence, $K_1 + K_2 = K$. Our initial estimation is that K_1 is as close as possible to K_2; however, adjustments might be needed if the workload of the application cannot be equally distributed among the two sets of processors. For example, if one of application tasks is much more intensive that the other tasks, the partition containing that task will have a much larger workload and, hence K_1 and K_2 have to be adjusted.

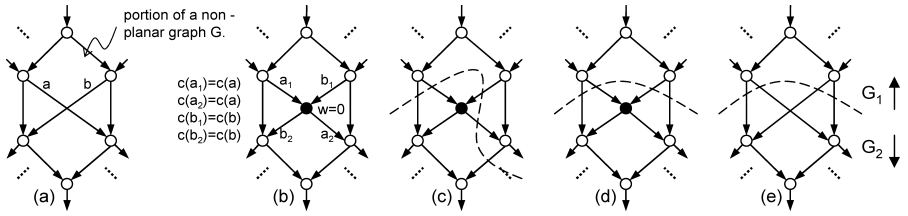


Fig. 9. (a) Portion of a sample non-planar graph G with one edge crossing. (b) Transforming G into planar graph G_p by adding a dummy node. (c) An invalid cut in G_p. (d) A valid cut in G_p. (e) The resulting cut in G.

The bipartitioning algorithm of Section 5 can be used in many different ways; however, here, we use it to develop only a one-dimensional K-way heuristic algorithm as follows. The cost function $Q_C = max(W_{G_1} + C_C, C_C + W_{G_2})$ is replaced with $Q_C = max(\frac{W_{G_1}}{K_1} + C_C, C_C + \frac{W_{G_2}}{K_2})$, where $K_1 = 1$ and $K_2 = 1$ in the original $K = 2$ case. Therefore, the cost function remains intact for bipartitioning case.

For $K > 2$, we try to consider the effect of future partitions. As graph G is partitioned into two subgraphs G_1 and G_2, we take into account that G_1 and G_2 will be partitioned into K_1 and K_2 subgraphs, respectively. That's why in the new cost function, W_{G_1} is divided by K_1 and W_{G_2} by K_2. However, the communication latency (C_C) term is not changed in the updated cost function, because it should represent the cost of communication between processor K_1 and $K_1 + 1$ in the pipeline.

7.4 Task Graph Planarization

In our discussions so far, we assumed that application task graph is planar. Some programming languages, such as StreamIt [Thies et al. 2002], guarantee the planarity of specified applications. However, our proposed method does not require developers to use a specific programming language, and thus, the input graph might be nonplanar. We introduce a transformation to *planarize* nonplanar task graphs, and to make them amenable to our task assignment technique.

For a given nonplanar input graph G, we start with the best embedding of G with minimum number of edge crossings. The goal is to eliminate edge crossings by adding dummy nodes at every edge crossing while guaranteeing that graph partitioning estimations are accurate. We refer to the planarized graph as G_p. As an example, Figure 9(a) shows a portion of a nonplanar graph with one crossing, and Figure 9(b) is the planarized graph with one dummy node.

Conceptually, the dummy node passes data from its incoming edges to corresponding outgoing edges (i.e., data on edge a_1 goes to edge a_2 and the data on b_1 goes to b_2). However, the node and its corresponding computation do not appear in the generated code. We assign computation workload and communication cost values to the dummy bypass node and introduced edges, so the G_p can be correctly analyzed using our algorithm. We show that convex cuts of G_p correspond to convex task assignments in G.

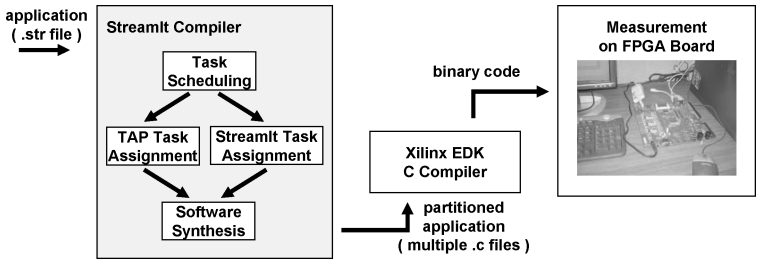


Fig. 10. Flow of experiments carried out to validate our task assignment technique (TAP)

The introduced edges in G_p have the same communication latencies as their corresponding edge in the original nonplanar graph G. Specifically, $c(a_1) = c(a_2) = c(a)$ and $c(b_1) = c(b_2) = c(b)$ (Figure 9(b)). In addition, workload of the dummy node is set to zero ($w(dummy) = 0$) because dummy node does not introduce any additional computation in the synthesized software. It will be removed before code generation by carefully reordering data communication during code generation.

Note that a convex cut in G_p gives a convex task assignment in G. Furthermore, the computation workload and communication latency of such a convex cut in G_p accurately models the workload and interprocessor communication of the corresponding task assignment in G. A convex cut in G_p does not cross both a_1 and a_2. Crossing both a_1 and a_2 implies nonconvexity (Figures 9(c) and 9(d)) and is not considered in this article (Section 4). As a result, we partition G_p without crossing the same edge of G twice, which enables us to infer a valid task assignment solution after partitioning G_p (Figure 9(e)).

8. EXPERIMENTAL PLATFORM

Our evaluation is based on *measurements* of application throughput on actual hardware. We use Digilent XUP Virtex-II PRO FPGA board to prototype single-, dual- and multi-core architectures. Xilinx MicroBlaze soft processors are used as processing resources. 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. Interprocessor FIFO communication channel is implemented using Xilinx 32-bit fast simplex links (FSL) with buffer size of 256 words. Processors and FSL both run at 100MHz. Figure 10 summarizes the experiments that we carried out to validate TAP.

We utilize MIT StreamIt 2.1 [Gordon et al. 2002] compiler framework to evaluate our algorithms. StreamIt is a programming language whose semantics are closely related to synchronous dataflow model of computation [Lee and Messerschmitt 1987b] with a few enhancement. Specifically, standard SDF model is enhanced to allow application initialization phase and utilization of limited control flow in program specification. In addition, StreamIt provides

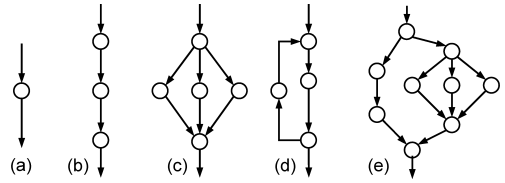


Fig. 11. (a) StreamIt Filter (b) Pipeline (c) SplitJoin (d) FeedbackLoop (e) Example task graph structure in StreamIt

an open-source compilation framework for stream programs specified in its language.

StreamIt compiler takes as input an application specified in enhanced synchronous dataflow 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 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 2.1 compiler framework 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.

8.1 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 (Figure 11).

Pipeline implements a chain of filters in which, a node gets its input from previous node and passes its output to next node. SplitJoin specifies independent data-parallel or task-parallel streams that diverge from a common splitter and merge into a common joiner. SplitJoin is similar in spirit to scatter-gather operator in parallel computing domain. FeedbackLoop also has a splitter and a joiner, but the joiner appears first in dataflow to join input with the output of feedback path.

Application graphs that are hierarchically composed using Pipelines, SplitJoins, and FeedbackLoop composition rules belong to class of series-parallel graphs, and thus, are planar by construction. This implies that many existing streaming applications are, or can be, modeled as planar task graphs. Note that for any application that cannot be specified as a planar task graph, the transformation discussed in Section 7.4 can safely transform the input task graph to a planar graph.

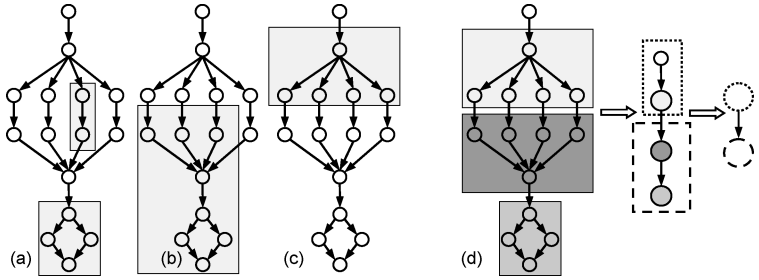


Fig. 12. (a) Two sample subgraphs that are candidate for Fusion. (b) A subgraph which is not considered as a Fusion candidate. (c) A Fusion candidate which is allowed to cut through a SplitJoin. (d) A possible series of hierarchical Fusions for task assignment to a target with two processors.

8.2 Task Assignment Algorithm in StreamIt 2.1

There are two separate situations for which StreamIt 2.1 provides two separate algorithms [Gordon et al. 2002]. In the first case, the number of available processors are more than the number of tasks (nodes) in the application task graph. Therefore, the compiler employs a series of `Fission` transformations in order to split tasks across multiple processors. Specifically, this algorithm looks for stateless tasks and duplicates them in order to explore data parallelism.

However, in most platforms, the number of tasks is more than the number of available processors. In this second case, the compiler applies a series of `Fusion` transformations. Tasks are grouped together so that the resulted task graph has the same number of tasks as processors. This is similar to our approach, that is, the method is a graph partitioning algorithm which explores task parallelism.

The partitioning algorithm tries to balance the computation workload across processors. It hierarchically fuses subgraphs until the task graph fits on the target architecture [Thies et al. 2003]. However, not all possible subgraphs are considered as candidates for Fusion. The algorithm considers only the subgraphs that are structured, that is, hierarchically composed based on the three composition rules of Figure 11. For example, the two sample subgraphs shown in Figure 12(a) are candidate for Fusion, but the subgraph shown in Figure 12(b) is not.

Therefore, the graph partitioning is restricted to certain cuts. As a result of the abovementioned restriction on subgraphs, a cut does not pass through a SplitJoin construct. However, if a cut falls at the same position on all branches of the SplitJoin, the algorithm does consider that cut. Figure 12(c) shows such situation.

Note that in [Gordon et al. 2002] a heuristic algorithm is presented to explore both task and data parallelism at the same time. It also employs software pipelining at task level in order to run pipelined tasks in parallel. For evaluation purposes, we compare against the latest publicly available StreamIt framework which uses the original algorithm [Gordon et al. 2002].

Application	Description	Task Graph		Throughput (outputs per sec.)
		V	E	
MATMUL	Blocked Matrix Multiply	23	23	134,700
FFT	Fast Fourier Transform	82	106	318,300
TDE	Freq. Domain Convolution	26	28	578,100
FILTER	Discrete Filter	29	31	17,900
BSORT	Bitonic Sort	314	407	258,100
DCT	Discrete Cosine Transform	14	19	102,700

Fig. 13. Benchmark applications. V and E columns show number of vertices and edges in task graph of each application. Last column is measured throughput of single-core hardware.

8.3 Computation Workload and Inter-Task Communication Estimation

We profiled MicroBlaze processor to estimate its cycle per instruction (CPI) distribution. Subsequently, internal computations of tasks 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 in 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 [Lee and Messerschmitt 1987a]. 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 $w_v = n(v) \times w'(v)$, 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 interprocessor channel.

9. RESULTS

9.1 Performance of Exact Algorithm

Our first experiment results, compares applications throughput using StreamIt 2.1 and our exact task assignment (TAP) algorithms. Figure 13 shows different benchmarks used in our evaluations. They are well-known stream applications which are building blocks of many embedded platforms such as multimedia and signal processing. The applications are selected from the StreamIt 2.1 benchmark set, having in mind the data and instruction memory constraints of our FPGA board.

Throughput of above benchmarks is measured on operating hardware to verify the effectiveness of our proposed approach. First, it is measured for a single-core hardware as a baseline for comparisons. This enables us to compare

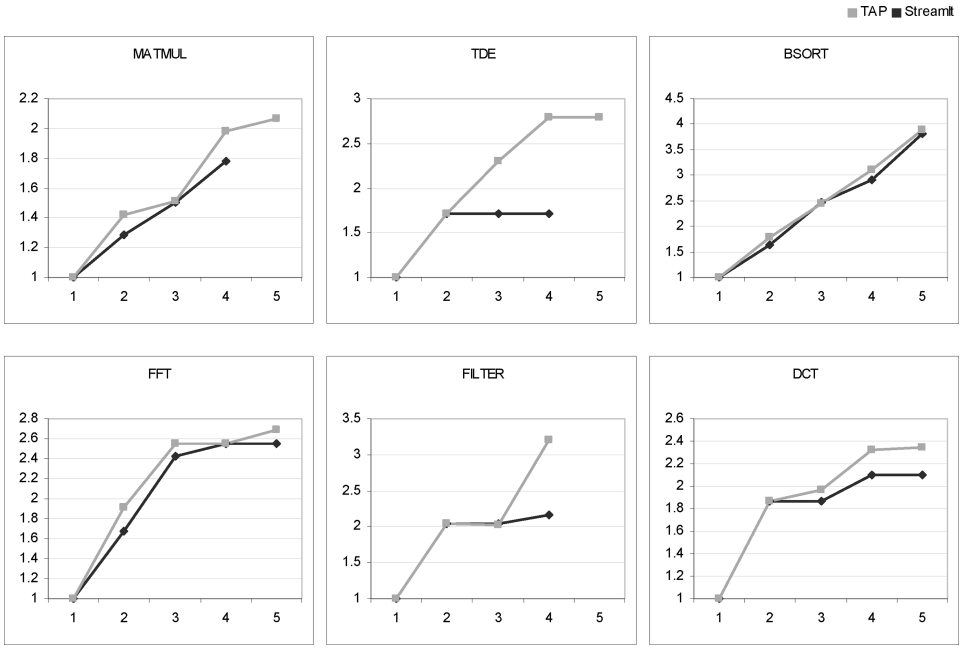


Fig. 14. Throughput of the multicore hardware with $K = 2$, 3, 4 and 5 processors, normalized by the single-core baseline from Figure 13, for both StreamIt 2.1 and TAP partitioning algorithms. X axis is the number of processors and Y axis is the normalized throughput. We use the exact algorithm (Section 5) for $K = 2$, and the heuristic approach (Section 7.3) for $K \geq 3$.

throughput of a K-core hardware with its ideal peak (i.e., K times the single-core throughput).

The last column of Figure 13 shows the single-core results. Throughput is measured as the number of outputs (data samples) per second produced by the hardware. We can convert these values to Byte/sec. by multiplying them by a factor of 4 because every data sample is either an `int` or a `float` number, and in our MicroBlaze hardware, they are both four bytes long.

Figure 14 shows measured throughput of the multicore hardware for $K = 2$, 3, 4, and 5 processors, normalized by the single-core baseline from Figure 13. Results are shown for both StreamIt 2.1 and TAP algorithms. For $K = 2$, we use the optimal task assignment algorithm described in Section 5, but for $K \geq 3$, we use the heuristic approach of Section 7.3. The figure shows that throughput does not linearly scale with the number of processors (i.e., after some point reaches saturation). This is the case in any multiprocessor system, mainly due to the overhead of interprocessor communications. Previous works by StreamIt group on 4×4 array of RAW cores show similar behavior Gordon et al. [2002, 2006].

Figures 15, 16, and 17 present more analysis on Figure 14 data. Figure 15 shows the amount of extra throughput gained by TAP comparing to StreamIt 2.1:

$$100 \times \frac{Throughput(TAP) - Throughput(StreamIt)}{Throughput(single - core)}$$

	MATMUL	FFT	TDE	FILTER	BSORT	DCT	Avg.
2-core	14	25	0	0	15	0	9.0
3-core	1	13	59	-2	-2	11	13.3
4-core	20	0	109	104	20	22	45.8
5-core	-	13	-	-	6	24	14.3

Fig. 15. Percentage of extra throughput gained by TAP comparing to StreamIt 2.1, for $K = 2$, 3, 4, and 5 processors. Note that this data is not a comparison with baseline.

For two processors, TAP has 25%, 15%, and 14% more throughput on FFT, BSORT, and MATMUL, and no extra gain on the other three benchmarks. This reconfirms our theoretical claim of optimality for exact task assignment algorithm of Section 5 (i.e., the algorithm is always better than [or equal with] another method, which is StreamIt 2.1 in this case).

As shown in Figure 15 for $K \geq 3$, out of 18 different cases (3×6), the heuristic method gains an extra throughput of more than 100% on two cases. However, as shown in the figure, it may sometimes lead to throughput degradation. This is expected because there is no theoretically proven result on the optimality of heuristic methods in general.

Figure 16 shows workload distribution of TAP and StreamIt 2.1 task assignment algorithms. For example, it shows that StreamIt has assigned more than 50% of the total computation workload of MATMUL to fourth processor of the quad-core hardware ($p4$ in top left of Figure 16(b)). This is because MATMUL has a compute intensive node in its task graph which has a heavy workload.

The cost function in Section 5 does not directly minimize the workload imbalance, but Figure 16 shows that TAP also balances the computation workload better than StreamIt. In some cases, the hierarchical graph structure of benchmark applications does not allow StreamIt to effectively partition the task graph, but our graph bipartitioning algorithm is not limited to the hierarchical structure and thus has a larger search space. Figure 17 shows that this case happens for TDE application.

By comparing Figure 15 and 16, we see that system throughput does not directly correlate to how well the workload is distributed. For example, in MATMUL with $K = 2$ cores, TAP increases the workload imbalance, but it actually achieves an 14% higher throughput. This is mainly because throughput is an implementation-dependent function of both computation workloads and communication latencies.

9.2 Comparing Exact and Approximate Algorithms

Recall that runtime and memory requirement of TAP depend on the value of estimated workload. TAP's runtime and memory demand is quite reasonable for small benchmarks. Figure 18 shows compilation time of TAP and StreamIt algorithms.

However, since the algorithm is pseudopolynomial not strictly polynomial, both runtime and memory requirement grow exponentially. 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

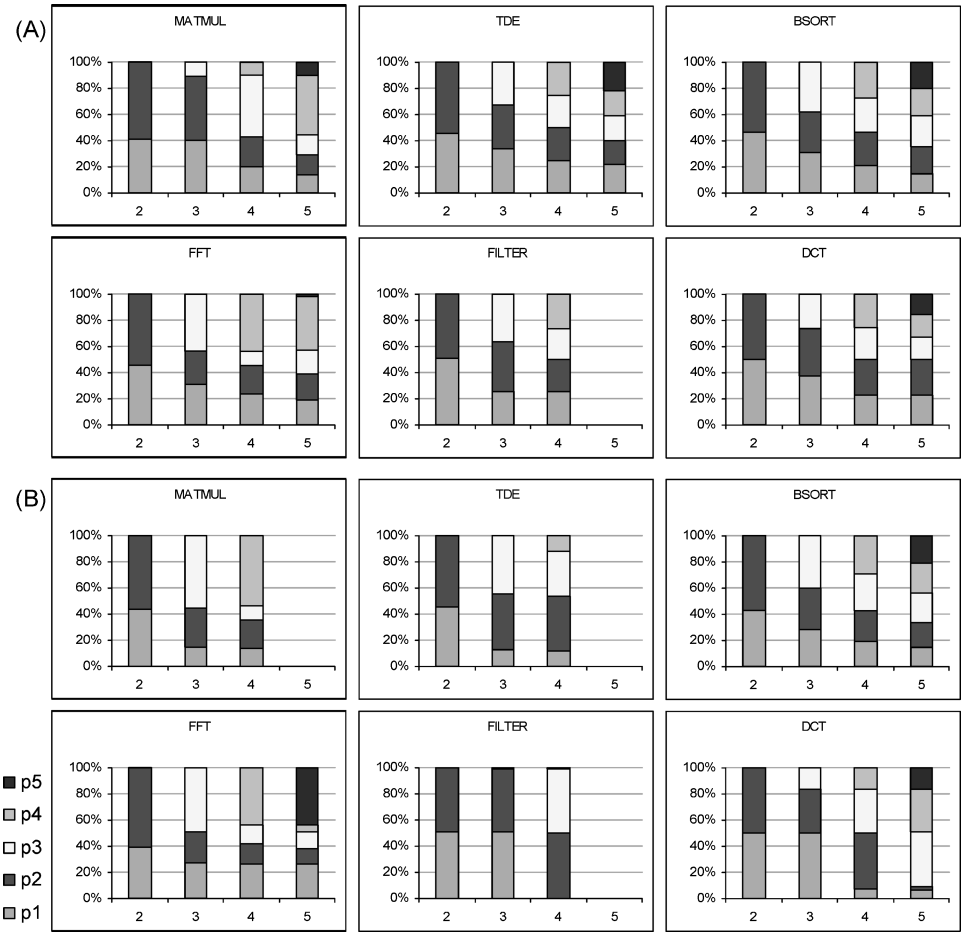


Fig. 16. Workload distribution for $K = 2$, 3, 4, and 5 processors with (a) TAP and (b) StreamIt 2.1 task assignment algorithm. X axis is the number of processors and Y axis is the workload distribution. Each color represents one processor of the multicore hardware.

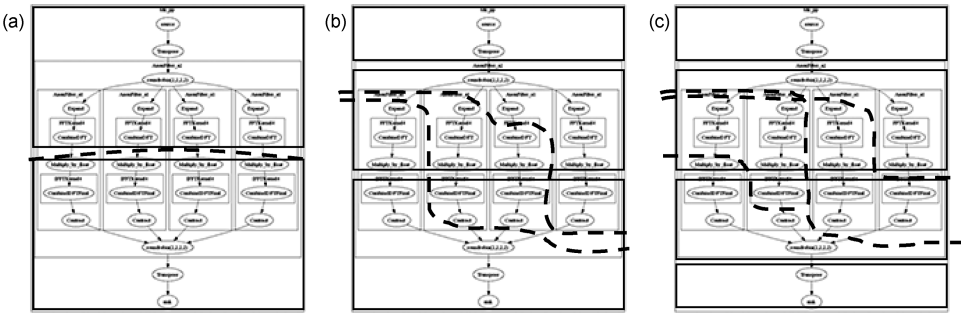


Fig. 17. Task assignment for TDE with (a) 2, (b) 3 and (c) 4 processors. Dotted lines are for TAP, and boxes are for StreamIt 2.1.

	2-core		3-core		4-core		5-core	
	StreamIt	TAP	StreamIt	TAP	StreamIt	TAP	StreamIt	TAP
MATMUL	229	134	289	240	272	181	309	195
FFT	1046	117	1145	153	1383	152	1401	173
TDE	319	50	340	91	314	92	329	101
FILTER	440	49	444	77	414	80	429	84
BSORT	3889	228	4497	315	5500	324	6213	341
DCT	172	15	171	35	163	35	202	38

Fig. 18. Runtime of StreamIt 2.1 and TAP task assignment algorithms in milliseconds.

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

Fig. 19. Modified benchmark applications. V, E, and F columns show number of vertices, edges, and faces in the task graph of each application.

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

Fig. 20. Runtime and memory requirement for exact task assignment algorithm.

estimation preserves their relative intensity, and ideally, should not affect task assignment quality.

Note that this experiment is done only for a dual-core architecture. Since now there is more memory on the FPGA board for each processor, we modified the testbenches and increased their computation requirement (Figure 19).

Figure 20 shows the time and memory required to run the exact task assignment algorithm, along with its 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 53-only 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 2.5GB of memory, which impedes its utilization in many systems.

Subsequently, we apply our approximate task assignment algorithm to trade throughput for algorithm time and memory requirement. Figure 21 shows the same parameters as Figure 20 but for the near-optimal solution offered by the approximate algorithm. Throughput, runtime and memory values are normalized with respect to their values from Figure 20. For example, in BSORT with

	Epsilon	Runtime (%)	Memory (%)	Throughput (%)
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
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

Fig. 21. Normalized runtime, memory and throughput for selected approximation bounds.

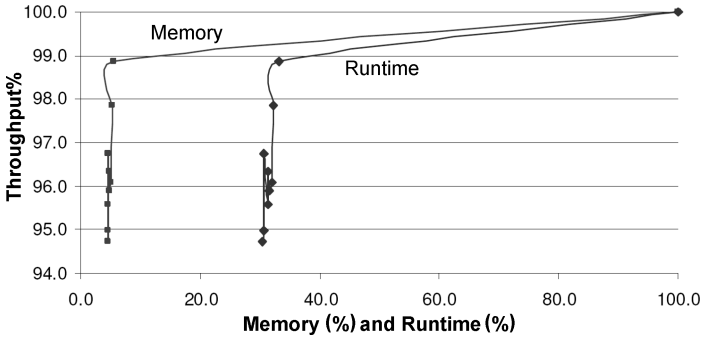


Fig. 22. Geometric mean of throughput degradation versus runtime improvement.

$\epsilon = 0.1$, 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 space 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 22 visualizes the geometric mean of throughput-memory and throughput-runtime tradeoff points, over all applications and all ϵ values (0.1, 0.2, ... 0.9). Note that Figure 21 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.

10. CONCLUSIONS

In this article, we presented a methodology for synthesizing embedded streaming applications for pipelined execution on multicore architectures. We first developed an exact, that is, theoretically optimal, bipartitioning algorithm called TAP, which jointly considers both the computation workload assigned to cores and inter-core communication traffics. We discussed how the algorithm is capable of optimizing realistically arbitrary function of the two factors. We also presented an approximation of the algorithm in order to have strictly linear runtime. In addition, we deigned a heuristic method for three or more processor cores based on dual-core TAP algorithm. We measured the actual application throughput on real multicore architectures. Hardware measurements (not simulations) showed that our exact algorithm for two cores yields 1.7× throughput improvement over single core baseline. In addition, our approximation method offers a range of runtime-throughput tradeoff points. For example, it runs about 3 times faster and requires about 20 times less memory, while results in throughput degradation of only 1% to 5%.

ACKNOWLEDGMENTS

We wish to thank all the reviewers for their constructive comments and we would also like to thank Adam Harbour for helping with the experiments.

REFERENCES

ALEKSANDROV, L., DJIDJEV, H., GUO, H., AND MAHESHWARI, A. 2007. Partitioning planar graphs with costs and weights. *J. Exper. Algor.* 11.

ALUR, R. 2003. Formal analysis of hierarchical state machines. In *Verifcation Theory and Practice*. Springer, Berlin, Germany, 42–66.

ALUR, R., COURCOUBETIS, C., HENZINGER, T. A., AND HO, P. H. 1992. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Proceedings of the 4th Annual Conference on Hybrid Systems*. Springer, Berlin, Germany, 209–229.

ANGELINI, P., DI BATTISTA, G., AND PATRIGNANI, M. 2007. Computing a minimum-depth planar graph embedding in O(n4) time. Lecture Notes in Computer Science, vol. 4619, 287.

ATASU, K., POZZI, L., AND IENNE, P. 2003. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the Design Automation Conference (DAC)*. IEEE, Los Alamitos, CA, 256–261.

BALARIN, F., WATANABE, Y., HSIEH, H., LAVAGNO, L., PASSERONE, C., SANGIOVANNI-VINCENTELLI, A. 2003. Metropolis: An integrated electronic system design environment. *IEEE Comput. 36,* 4, 45–52.

BENVENISTE, A., CARLONI, L. P., CASPI, P., AND SANGIOVANNI-VINCENTELLI, A. L. 2003. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*. Springer, Berlin, Germany, 35–50.

BONIVENTO, A., CARLONI, L. P., AND SANGIOVANNI-VINCENTELLI, A. L. 2005. Rialto: A bridge between description and implementation of control algorithms for wireless sensor networks.

In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT)*. Springer, Berlin, Germany, 183–186.

BUI, T. N. AND PECK, A. 1992. Partitioning planar graphs. *SIAM J. Comput. 21*, 2, 203–215.

CONG, J., HAN, G., AND JIANG, W. 2007. Synthesis of an application-specific soft multiprocessor system. In *Proceedings of the 15th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, New York, 99–107.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

ERBAS, C., ERBAS, S. C., AND PIMENTEL, A. D. 2006. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Trans. Evolut. Comput. 10,* 3, 358–374.

FEIGE, U. AND KRAUTHGAMER, R. 2002. A polylogarithmic approximation of the minimum bisection. *SIAM J. Comput. 31*, 4, 1090–1118.

GAREY, M. R. AND JOHNSON, D. S. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York.

GARG, N., SARAN, H., AND VAZIRANI, V. V. 2000. Finding separator cuts in planar graphs within twice the optimal. *SIAM J. Comput. 29,* 1, 159–179.

GORDON, M. I., THIES, W., AND AMARASINGHE, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. ACM, New York, 151–162.

HENZINGER, T. A., MANNA, Z., AND PNUELI, A. 1992. Towards refining temporal specifications into hybrid systems. In *Proceedings of the 5th International Conference on Hybrid Systems*. Springer, Berlin, Germany, 60–76.

HENZINGER, T. A., NICOLLIN, X., SIFAKIS, J., AND YOVINE, S. 1994. Symbolic model checking for real-time systems. *Inform. Comput. 111*, 2, 193–244.

HENZINGER, T. A., QADEER, S., AND RAJAMANI, S. K. 1998. You assume, we guarantee: Methodology and case studies. In *Proceedings of the 10th International Conference on Computer Aided Verification*. Springer, Berlin, Germany, 440–451.

HENZINGER, T. A. AND SIFAKIS, J. 2006. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods*. Springer, Berlin, Germany, 1–15.

HU, J. AND MARCULESCU, R. 2005. Energy- and performance-aware mapping for regular noc architectures. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst. 24,* 4.

KAHN, G. 1974. The semantics of simple language for parallel programming. In *Proceedings of the International Federation for Information Processing (IFIP) Congress*. 471–475.

KARPINSKI, M. 2002. Approximability of the minimum bisection problem: An algorithmic challenge. In *Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science (MFCS'02)*. Springer, Berlin, Germany, 59–67.

LEE, E. A. 2005. Building unreliable systems out of reliable components: The real time story. Tech. rep. UCB/EECS-2005-5, EECS Department, University of California, Berkeley.

LEE, E. A. 2006. The problem with threads. *IEEE Comput. 39*, 5, 33–42.

LEE, E. A. and MESSERSCHMITT, D. G. 1987a. Static scheduling of synchronous data ow programs for digital signal processing. *IEEE Trans. Comput. 36*, 1, 24–35.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987b. Synchronous data ow. *Proc. IEEE 75,* 9, 1235–1245.

LIPTON, R. J. AND TARJAN, R. E. 1979. A separator theorem for planar graphs. *SIAM J. Applied Mathematics 36,* 177–189.

MA, Z., CATTHOOR, F., AND VOUNCKX, J. 2005. Hierarchical task scheduler for interleaving subtasks on heterogeneous multiprocessor platforms. In *Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC)*. IEEE, Los Alamitos, CA, 952–955.

MEETING. 2006. Joint United States-European Union-TEKES workshop: Long term challenges in high con_dence composable embedded systems. http://www.truststc.org/euus/wiki/Euus/HelsinkiMeeting.

MICHAEL I. GORDON, WILLIAM THIES, MICHAL KARCZMAREK, JASPER LIN, ALI S. MELI, ANDREW A. LAMB, CHRIS LEGER, JEREMY WONG, HENRY HOFFMANN, DAVID MAZE, AND SAMAN AMARASINGHE. 2002. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X). ACM, New York, 291–303.

OWENS, J. D. ET AL. 2000. Polygon rendering on a stream architecture. In *Proceedings of the Workshop on Graphics Hardware.* ACM, New York, 23–32.

OWENS, J. D. ET AL. 2002. Media processing applications on the Imagine stream processor. In *Proceedings of the IEEE/ACM International Conference on Computer Design (ICCD).* IEEE, Los Alamitos, CA, 295–302.

PARK, J. K. AND PHILLIPS, C. A. 1993. Finding minimum-quotient cuts in planar graphs. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC).* ACM, New York, 766–775.

PIMENTEL, A. D. ET AL. 2001. Exploring embedded-systems architectures with artemis. *IEEE Comput. 34*, 11, 57–63.

PINO, J. L., HA, S., LEE, E. A., AND BUCK, J. T. 1995. Software synthesis for DSP using ptolemy. *J. VLSI Signal Process. Syst. 9*, 1-2, 7–21.

PINTO, A., BONIVENTO, A., SANGIOVANNI-VINCENTELLI, A. L., PASSERONE, R., AND SGROI, M. 2006. System-level design paradigms: Platform-based design and communication synthesis. *ACM Trans. Des. Autom. Electron. Syst. 11*, 3, 537–563.

RANGAN, R., VACHHARAJANI, N., STOLER, A., OTTONI, G., AUGUST, D. I., AND CAI, G. Z. N. 2006. Support for high-frequency streaming in CMPs. In *Proceedings of the 39th Annual International Symposium on Microarchitecture.* IEEE, Los Alamitos, CA, 259–272.

RAO, S., AMIR, E., AND KRAUTHGAMER, R. 2003. Constant factor approximation of vertex-cuts in planar graphs. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. ACM, New York, 90–99.

RAO, S. B. 1992. Faster algorithms for finding small edge cuts in planar graphs. In *Proceedings of the ACM Symposium on Theory of Computing (STOC).* ACM, New York, 229–240.

STANKOVIC, J. A. 2007. Keynote speech: Control challenges in wireless sensor networks. In *Proceedings of the 10th International Conference on Hybrid Systems: Computation and Control*. Springer, Berlin, Germany, 2.

STUIJK, S. AND BASTEN, T. 2008. Analyzing concurrency in streaming applications. Kluwver *J. Syst. Architec.* (available online).

STUIJK, S., BASTEN, T., GEILEN, M., AND CORPORAAL, H. 2007. Multiprocessor resource allocation for throughput-constrained synchronous dataow graphs. In *Proceedings of the 44th Design Automation Conference (DAC)*. IEEE, Los Alamitos, CA, 777–782.

STUIJK, S., GEILEN, M., AND BASTEN, T. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataow graphs. In *Proceedings of the 43rd Design Automation Conference (DAC)*. IEEE, Los Alamitos, CA, 899–904.

SZTIPANOVITS, J., GLOSSNER, C. J., MUDGE, T. N., ROWEN, C., SANGIOVANNI-VINCENTELLI, A. L., WOLF, W., AND ZHAO, F. 2005. Panel session: Grand challenges in embedded systems. In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT)*. IEEE, Los Alamitos, CA, 333.

TAYLOR, M. B., PSOTA, J., SARAF, A., SHNIDMAN, N., STRUMPEN, V., FRANK, M., ET AL. 2004. Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Los Alamitos, CA, 2.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*. Springer, Berlin, Germany, 179–196.

THIES, W., LIN, J., AND AMARASINGHE, S. 2003. Partitioning a structured stream graph using dynamic programming. Tech. rep., CS Department, Massachusetts Institute of Technology.

THOEN, F. AND CATTHOOR, F. 2000. *Modeling, Verification, and Exploration of Task-Level Concurrency of Real-Time Embedded Systems*. Kluwer Academic Publishers.

YU, J., YAO, J., BHUYAN, L., AND YANG, J. 2007. Program mapping onto network processors by recursive bipartitioning and refining. In *Proceedings of the 44th Annual IEEE/ACM Design Automation Conference (DAC'04).* IEEE, Los Alamitos, CA, 805–810.

YU, Z., MEEUWSEN, M., APPERSON, R., SATTARI, O., LAI, M., WEBB, J., WORK, E., MOHSENIN, T., SINGH, M., AND BAAS, B. M. 2006. An asynchronous array of simple processors for DSP applications.

In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, Los Alamitos, CA.

ZHOU, G., LEUNG, M.-K., AND LEE, E. A. 2007. A code generation framework for actor-oriented models with partial evaluation. In *Proceedings of the International Conference on Embedded Software and Systems*. ACM, New York, 786–799.

Received August 2007; revised March 2008; accepted August 2008