Comparability Graph Coloring for Optimizing Utilization of Software-Managed Stream Register Files for Stream Processors [1]

Xuejun Yang
School of Computer, National University of Defense Technology
Li Wang
School of Computer, National University of Defense Technology
Jingling Xue
School of Computer Science and Engineering, University of New South Wales
and
Qingbo Wu
School of Computer, National University of Defense Technology

The stream processors represent a promising alternative to traditional cache-based general-purpose processors in achieving high performance in stream applications (media and some scientific applications). In a stream programming model for stream processors, an application is decomposed into a sequence of kernels operating on streams of data. During the execution of a kernel on a stream processor, all streams accessed must be communicated through a non-bypassing software-managed on-chip memory, the SRF (Stream Register File). Optimizing utilization of the scarce on-chip memory is crucial for good performance. The key insight is that the interference graphs (IGs) formed by the streams in stream applications tend to be comparability graphs or decomposable into a set of comparability graphs. We present a compiler algorithm for finding optimal or near-optimal colorings, i.e., SRF allocations in stream IGs, by computing a maximum spanning forest of the sub-IG formed by long live ranges, if necessary. Our experimental results validate the optimality and near-optimality of our algorithm by comparing it with an ILP solver, and show that our algorithm yields improved SRF utilization over the First-Fit bin-packing algorithm, the best in the literature.

1. INTRODUCTION

Hardware-managed cache has traditionally been used to bridge the ever-widening performance gap between processor and memory. Despite this great success, some deficiencies with cache are well-known. First, their complex hardware logic incurs

[1]**Extension of Conference Paper**. This journal paper has extended our earlier PPoPP'09 [Yang et al. 2009] in four directions:

—A general algorithm that works for any arbitrary stream IG is presented while our earlier algorithm is limited to stream IGs decomposable into comparability graphs and a forest.

—More benchmarks and more experimental evaluation are included.

—A counter example is presented showing First-Fit may occasionally achieve better colorings than our algorithm.

—All results are now rigorously proved.

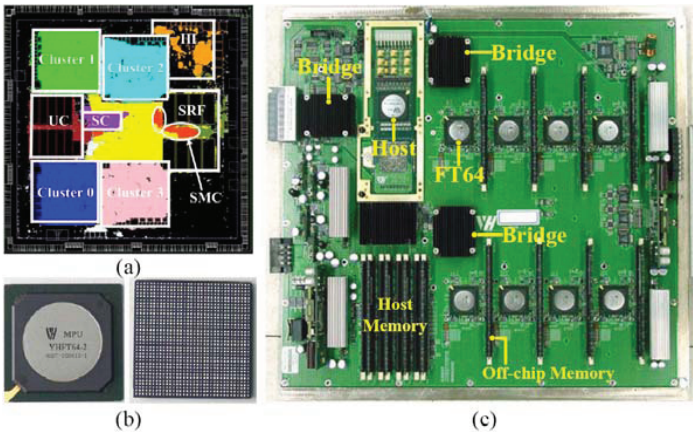
high overhead in power consumption and area. Second, their simple application-independent management strategy does not benefit from some data access characteristics in many applications. For example, media applications and some scientific applications exhibit producer-consumer locality with little global data reuse, which are hardly fully exploited by hardware-managed cache. Finally, their uncertain access latencies make it difficult to guarantee real-time performance.

In contrast to cache, software-managed on-chip memory has advantages in area, cost, and access speed, etc [Banakar et al. 2002]. It is thus widely adopted in embedded systems (known as scratchpad memory or SPM for short), stream architectures (known as stream register file, local memory or streaming memory), and GPUs (known as shared memory in NVIDIA's new generation GPUs under its CUDA programming model). In the case of supercomputers, software-managed on-chip memory is also frequently used, especially in their accelerators. Examples include Merrimac [Dally et al. 2003], Cyclops64 [Cuvillo et al. 2005], Grape-DR [Makino et al. 2007], Roadrunner [Koch 2006], and TianHe-1A (world's fastest supercomputer in TOP500 list released in November 2010).

The (programmable) stream processors, such as Imagine [Owens et al. 2002], Raw [Taylor et al. 2002], Cell [Williams et al. 2006], Merrimac [Dally et al. 2003] and GPUs, represent a promising alternative in achieving high performance in media applications. In addition, stream processing is also well suited for some scientific applications [Dally et al. 2003; Williams et al. 2006; Yang et al. 2007]. In [Yang et al. 2007], we introduced the design and fabrication of FT64, the first 64-bit stream processor for scientific computing. Like Imagine [Owens et al. 2002], Cell [Williams et al. 2006] and Merrimac [Dally et al. 2003], FT64, as shown in Figure 1, can be easily mapped to the stream virtual machine architecture described in [Labonte et al. 2004]. Such stream processor executes applications that have been mapped to the stream programming model: a program is decomposed into a sequence of computation-intensive kernels that operate on streams of data elements. Kernels are compiled to VLIW microprograms to be executed on clusters of ALUs, one at a time. Streams are stored in a software-managed on-chip memory, called SRF (Stream Register File).

The stream programming models, Brook [Buck et al. 2004], CUDA, StreamC/KernelC [Das et al. 2006] and StreamIt [Thies et al. 2001], which facilitate locality exploitation and bandwidth optimization, have been proven to be useful for programming stream architectures [Das et al. 2006; Yang et al. 2007; Kudlur and Mahlke 2008]. Some other research results also demonstrate their usefulness for general-purpose architectures [Gummaraju and Rosenblum 2005; Leverich et al. 2007; Gummaraju et al. 2008].

Research into advanced compiler technology for stream languages and architectures is still at its infancy. Among several challenges posed by stream processing for compilation [Das et al. 2006], a careful allocation of the scarce on-chip SRF becomes imperative. SRF, the nexus of a stream processor, is introduced to capture the widespread producer-consumer locality in media applications to reduce expensive off-chip memory traffic. Unlike conventional register files, however, SRF is non-bypassing, namely, the input and output streams of a kernel must be all stored in the SRF when a kernel is being executed. If the data set of a kernel is



(a) Chip layout. (b) Top and bottom views. (c) Basic Modules.

Fig. 1: FT64 stream processor.

too large to fit into the SRF, strip mining can be applied to segment some large streams into smaller strips so that the kernel can then be called to operate on one strip at a time. Alternatively, some streams can be double-buffered [Das et al. 2006] or spilled [Wang et al. 2008] until the data set of every kernel does not exceed the SRF capacity. Therefore, optimizing utilization of SRF is crucial for good performance. Presently, SRF utilization is predominantly optimized by applying First-Fit bin-packing heuristics [Das et al. 2006], which can be sub-optimal for some large applications.

In this paper, we present a new compiler algorithm for optimizing utilization of SRF for stream applications. The central machinery is the traditional interference graph (IG) representation except that an IG here is a weighted (undirected) graph formed by the streams operated on by a sequence of kernels. The key discovery is that the IGs in many media applications are comparability graphs, enabling the compiler to obtain optimal colorings in polynomial time. This has motivated us to develop a new algorithm for optimizing utilization of SRF when allocating the streams in stream IGs to the SRF by comparability graph coloring.

This paper makes the following main contributions:

—We show that stream IGs tend to be comparability graphs, which can thus be optimally colored.

—We propose to optimize utilization of SRF by comparability graph coloring and present a compiler algorithm for coloring arbitrary stream IGs through graph decompositions and maximal spanning forest computation, if necessary.

—We show by experiments that our algorithm can find optimal and near-optimal colorings efficiently for well-structured media and scientific applications that are amenable to stream processing, by comparing with both an ILP-based approach and a First-Fit based approach, thereby outperforming First-Fit heuristics.

The rest of this paper is organized as follows. For background information, Section 2 introduces the stream programming model and some graph theory results

to provide a basis for understanding our approach. Section 3 describes the SRF management problem we solve. Section 4 casts it as a comparability graph coloring problem and presents our algorithm for solving this new formulation. Section 5 evaluates our approach. Section 6 discusses related work. Section 7 concludes the paper.

2. BACKGROUND

2.1 Stream Programming Model

The stream programming model employed in FT64 is StreamC/KernelC, which is also used in the Imagine processor [Das et al. 2006]. The central idea behind stream processing is to divide an application into *kernels* and *streams* to expose its inherent locality and parallelism. As a result, an application is split into two programs, a *stream program* running on the host and a *kernel program* on the stream processor. The stream program specifies the flow of streams between kernels and initiates the execution of kernels. The kernel program executes these kernels, one at a time.

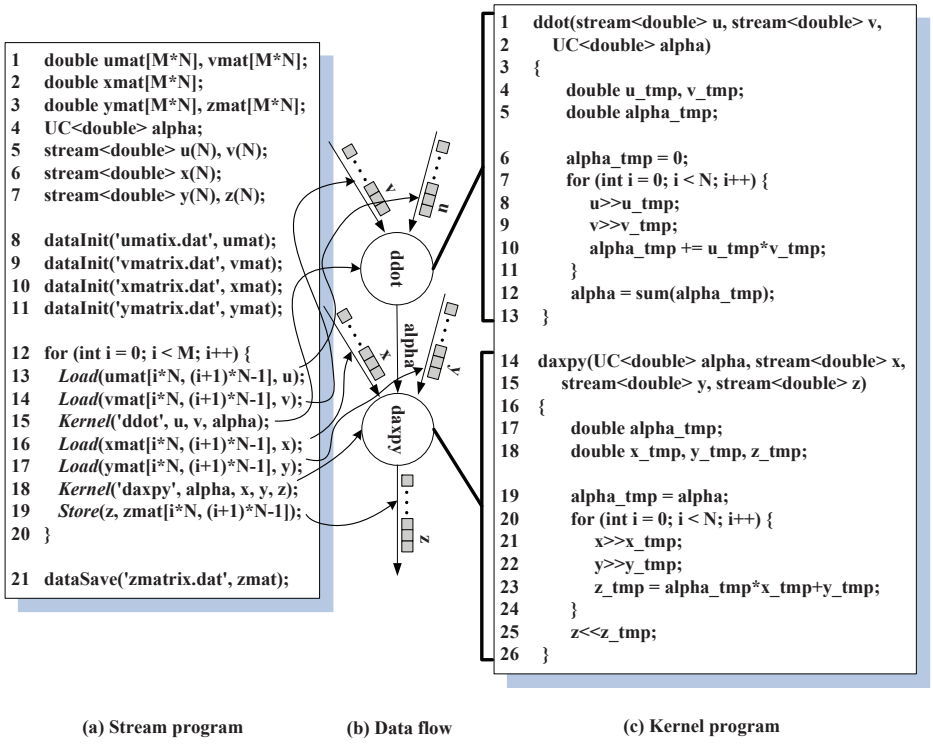


Fig. 2: Stream and kernel programs for a computation comprising *ddot* and *daxpy*.

Figure 2 depicts the mapping of a program comprising two BLAS kernels, *ddot* and *daxpy* to the stream programming model with FT64 as the underlying architecture. The program exhibits explicit producer-consumer locality: the output stream

from the last kernel execution is used as an input for the next kernel execution in sequence.

Consider the stream program in Figure 2(a) first. In lines 1 – 3, five arrays of size $M * N$ each are declared. In line 4, a UC variable (i.e., so-called microcontroller variable) is declared. In stream programming, UC variables are passed to a kernel loop as scalar arguments, which are often used in scientific algorithms as the coefficients of math equations or the results of vector reductions. In lines 5 – 7, five streams of size N each are declared. In lines 8 – 11, the function *dataInit* is called four times to initialize arrays *umat*, *vmat*, *xmat* and *ymat* residing in the off-chip memory with the four data files stored at the host. In line 13, the data from the current (loop dependent) section in *umat* are gathered into stream *u*. This will result in the loading of the data from *umat* in off-chip memory into the space allocated to stream *u* in the SRF. In line 14, stream *v* is initialized from array *vmat* similarly. In line 15, *ddot* is called to compute the dot product (inner product) of two double precision vectors represented by streams *u* and *v*. As shown, *u* and *v* are input streams and UC variable *alpha* is output. In lines 16 and 17, streams *x* and *y* are initialized from arrays *xmat* and *ymat*, respectively. In line 18, *daxpy* is called with *alpha*, *x* and *y* as input and *z* as output. After the kernel has run to completion, the final output stream is stored from the SRF back into array *zmat* in off-chip memory (line 19). In line 21, the result is saved into a data file.

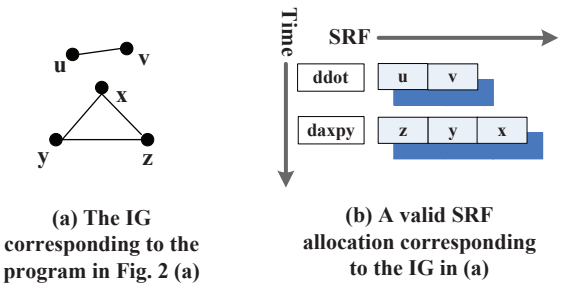
Consider the kernel program given in Figure 2(c), which is executed by FT64 in the VLIW mode. Let us examine *ddot* first. In line 7, a loop goes over each input stream. In line 8, four elements from stream *u* are read simultaneously at a time with each being assigned to a private temporary variable *u_tmp* on one of the four clusters in FT64 [Yang et al. 2007]. In line 9, the elements of stream *v* are read off similarly. In line 10, the computations on these elements are performed simultaneously on each cluster with the results being summed into a private temporary variable *alpha_tmp*. In line 12, the partial sums on four clusters are added up, with results being assigned to the UC variable *alpha*. For *daxpy*, the process is similar except that the results are appended to output stream *z*, four at a time.

A stream program consists of a sequence of loops where each loop includes a sequence of kernels operating on streams. In a stream compiler, all loops are considered separately in SRF allocation. For FT64 [Yang et al. 2007], the DRAM controller supports two stream-level instructions, *Load* and *Store*, that transfer an entire stream between off-chip memory and the SRF. In stream programs as demonstrated in Figure 2(a), loads and stores are used to initialize some streams from the global input data residing in off-chip memory and write certain streams to off-chip memory, respectively.

The central machinery in our approach to allocating the streams in a loop to the SRF is the traditional interference graph (IG) except that it is a weighted (undirected) graph formed by the streams operated on by the kernels in the loop. All streams accessed in the loop are identified as live ranges to be placed in the SRF. If two live ranges interfere (i.e., overlap), they must be placed in non-overlapping SRF spaces. The live ranges of streams are computed by extending the *def/use* definitions for scalars to streams: *Load* defines a stream, *Store* uses a stream, and a kernel call (re)defines its output streams and uses its input streams. The live range

of a stream starts from its definition and ends at its last use. To achieve better allocation results, streams are renamed using the SSA (static single assignment) form.

After the live ranges have been computed for a loop, its IG, denoted $\mathcal{G}$, is built in the normal manner, where a weighted node denotes a stream live range whose weight is the size of the stream and an edge connects two nodes if their live ranges interfere with each other.



(a) The IG corresponding to the program in Fig. 2 (a)

(b) A valid SRF allocation corresponding to the IG in (a)

Fig. 3: The interference graph and the allocation.

Consider the stream program in Figure 2 (a), its IG is depicted in Figure 3 (a), and a valid allocation is given in Figure 3 (b).

2.2 Interval Coloring and Comparability Graph

Section 2.2 recalls the basic results about interval coloring and comparability graph from [Golumbic 2004], which provide a basis for understanding our approach and proving its optimality and near-optimality.

2.2.1 *Basic Definitions.* Given a (directed or undirected) graph $\mathcal{G} = (V, E)$ and a subset $A \subseteq V$, the *induced subgraph* by A is $\mathcal{G}(A) = (A, E(A))$, where $E(A) = \{(x, y) \in E \mid x, y \in A\}$. A subset $A \subseteq V$ of r nodes is a *clique* or *r-clique* if it induces a complete subgraph. A clique is a *maximal clique* if it is not contained in any other clique.

Given an undirected graph $\mathcal{G} = (V, E)$ with positively integral node weights $w : V \to \mathbb{N}$, an *interval coloring* α of $\mathcal{G}$ maps each node x onto an interval I_x of a real line of width $w(x)$ so that adjacent nodes are mapped to disjoint intervals, i.e., $(x, y) \in E$ implies $I_x \cap I_y = \emptyset$. It is well-known that interval coloring is NP-complete [Garey and Johnson 1979]. The total *width* of an interval coloring α, $\chi_\alpha(\mathcal{G}; w)$, is $|\bigcup_{x \in V} I_x|$. The *chromatic number* $\chi(\mathcal{G}; w)$ is the smallest width used to color the nodes in $\mathcal{G}$. The *clique number* is:

$$\omega(\mathcal{G}; w) \;=\; \max\{w(K) \mid K \text{ is a clique of } \mathcal{G}\} \tag{1}$$

which is the weight of a heaviest clique.

As a result, the following relation always holds:

$$\chi(\mathcal{G}; w) \;\geqslant\; \omega(\mathcal{G}; w) \tag{2}$$

2.2.2 *Interval Coloring vs. Acyclic Orientation.* Let $\mathcal{G}=(V,E)$ be an undirected graph. An *orientation* of $\mathcal{G}$ is a function α that assigns every edge a direction such that $\alpha(x,y) \in \{(x,y),(y,x)\}$ for all $(x,y) \in E$. Let $\mathcal{G}_\alpha$ be the digraph obtained by replacing each edge $(x,y) \in E$ with the arc $\alpha(x,y)$. An orientation α is said to be *acyclic* if $\mathcal{G}_\alpha$ contains no directed cycles.

Every interval coloring α of $\mathcal{G}$ induces an acyclic orientation α' such that $(x,y) \in \alpha'$ if and only if I_x is to the right of I_y for all $(x,y) \in E$. Conversely, an acyclic orientation α of $\mathcal{G}$ induces an interval coloring α'. For a sink node x (without successors), let $I'_x = [0, w(x))$. Proceeding inductively, for a node y with all its successors already colored, let $I'_y = [t, t + w(y))$, where t is the largest endpoint of their intervals.

In an optimal coloring, the chromatic number $\chi(\mathcal{G}; w)$ is related to the notion of *heaviest path* in an acyclic orientation of $\mathcal{G}$ as follows:

$$\chi(\mathcal{G}; w) \;=\; \min_{\alpha \in \mathcal{A}(\mathcal{G})} \big(\max_{\mu \in \mathcal{P}(\alpha)} \mathrm{w}(\mu) \big) \tag{3}$$

where $\mathcal{A}(\mathcal{G})$ is the set of all acyclic orientations of $\mathcal{G}$, $\mathcal{P}(\alpha)$ the set of directed paths in an orientation $\alpha \in \mathcal{A}(\mathcal{G})$ and $\mathrm{w}(\mu)$ the total weight of the nodes of a directed path $\mu \in \mathcal{P}(\alpha)$. In other words, the orientation whose heaviest path is the smallest induces an optimal coloring. The heaviest-path-based formulation stated in (3) is exploited in the development of our coloring algorithm for stream IGs.

2.2.3 *Comparability Graph.* For the purposes of optimizing utilization of SRF, we examine below a class of graphs for which interval coloring can be found optimally in polynomial time.

Definition 2.1. An orientation α of an undirected graph $\mathcal{G}$ is *transitive* if $(x,z) \in \mathcal{G}_\alpha$ whenever $(x,y),(y,z) \in \mathcal{G}_\alpha$.

Definition 2.2. An undirected graph $\mathcal{G}$ is a *comparability graph* if there exists a transitive orientation of $\mathcal{G}$.

A transitive orientation is acyclic but not conversely, and there does not always exist a transitive orientation for an arbitrary graph. For example, a chordless cycle with an odd number of edges, as shown in Figure 4, is not a comparability graph.

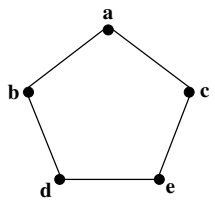


Fig. 4: A graph that is not a comparability graph.

Let α be a transitive orientation of a comparability graph $\mathcal{G}$. Due to transitivity, every path in $\mathcal{G}_\alpha$ is contained in a clique of $\mathcal{G}$. In particular, the heaviest path in $\mathcal{G}_\alpha$ equals to the heaviest clique in $\mathcal{G}$, i.e., $\chi(\mathcal{G}; w) \leqslant \chi_\alpha(\mathcal{G}; w) = \omega(\mathcal{G}; w)$. By further

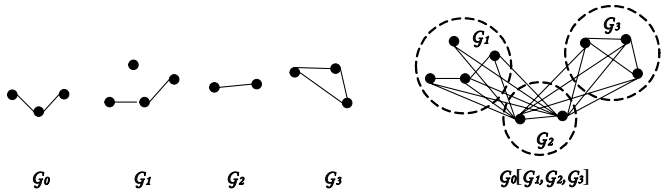


Fig. 5: An illustration of Definition 2.4 ($n = 3$).

applying (2), we conclude that $\chi_\alpha(\mathcal{G}; w) = \chi(\mathcal{G}; w) = \omega(\mathcal{G}; w)$ holds, as summarized below.

THEOREM 2.3. [Golumbic 2004] *For any transitive orientation α of $\mathcal{G}$, the interval coloring induced is optimal.*

Definition 2.4. Let $\mathcal{G}_0$ be a graph with n nodes $v_1, v_2, \ldots, v_n$ and $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ be n disjoint graphs. These graphs may be directed or undirected. The *composition* graph $\mathcal{G} = \mathcal{G}_0[\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n]$, which is illustrated pictorially in Figure 5, is formed formally as follows: First, replace v_i in $\mathcal{G}_0$ with $\mathcal{G}_i$. Second, for all $1 \leq i, j \leq n$, make each node of $\mathcal{G}_i$ adjacent to each node of $\mathcal{G}_j$ whenever v_i is adjacent to v_j in $\mathcal{G}_0$. Formally, for $\mathcal{G}_i = (V_i, E_i)$, we define $\mathcal{G} = (V, E)$ as follows:

$$V = \cup_{1 \leqslant i \leqslant n} V_i$$
$$E = \cup_{1 \leqslant i \leqslant n} E_i \cup \{(x, y) \mid x \in V_i, y \in V_j \text{ and } (v_i, v_j) \in E_0\}$$

THEOREM 2.5. [Golumbic 2004] *Let $\mathcal{G} = \mathcal{G}_0[\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n]$, where all $\mathcal{G}_i$'s are disjoint undirected graphs. Then $\mathcal{G}$ is a comparability graph if and only if all $\mathcal{G}_i$'s are comparability graphs.*

Furthermore, the problems of recognizing a comparability graph $\mathcal{G} = (V, E)$ and finding a transitive orientation of $\mathcal{G}$ can both be done in $O(\delta \cdot \mid E \mid)$ time and $O(\mid V \mid + \mid E \mid)$ space, where δ is the maximum degree of a node in $\mathcal{G}$. Based on α, an optimal coloring of $\mathcal{G}$ can be obtained in linear time [Golumbic 2004].

3. PROBLEM STATEMENT

This work focuses on optimizing utilization of the SRF. So only stream programs are relevant. Given a stream program, this paper presents an algorithm that assigns the streams in the program to the SRF so as to minimize the total amount of space taken by the streams. Such an algorithm can then be used by a stream compiler to produce a final SRF allocation by combining with live range splitting, if necessary.

The SRF allocation problem can be naturally solved as an interval-coloring problem as presented in Section 2.2, allocating SRF spaces to stream live ranges in an IG is represented by an assignment of intervals to the nodes in the IG, and minimizing the span of intervals amounts to minimizing the required SRF size.

Let us see why our comparability graph coloring based algorithm could achieve better SRF allocation than First-Fit, which is the approach adopted in the state-of-the-art stream compilers. First-Fit places the streams in an IG in a certain order. There are two popular choices, denoted *First-Fit-1* and *First-Fit-2*. First-Fit-1 processes streams in decreasing order of stream sizes, which is the heuristic

used in [Das et al. 2006]. First-Fit-2 processes streams according to when their live ranges begin and then what their stream sizes are. First-Fit heuristics are not sensitive to the structure of an IG thereby resulting in SRF fragmentation. We examine this with two simple programs shown in Figure 6(a) and Figure 7(a), with their dataflow graphs depicted in Figure 6(b) and Figure 7(b), respectively.

Let us consider the first example first. Figure 6(c) shows the SRF allocation for the program in Figure 6(a) under First-Fit-1. The streams S_2, S_1 and S_4 are allocated first because they are the heaviest, followed by S_6, S_5 and S_3, resulting in poor SRF utilization. In contrast, based on the IG and the assigned transitive orientation in Figure 6(f), the optimal SRF allocation found by our algorithm is shown in Figure 6(e). The gap between the two is 32 (15.4%) but can be larger in general. Let us consider the second example. Figure 7(c) shows the SRF allocation for the program given in Figure 7(a) under First-Fit-2. The streams S_1 and S_2 live at kernel 1 are allocated before S_3 and S_4. S_1, which is heavier than S_2, is allocated first followed by S_2. However, since S_2 is also live in kernel 2, S_3, which is heavier than S_1, can only be placed after S_2. Similarly, S_4, which is heavier than $S_1 + S_2$, can only be placed after S_3, resulting in SRF fragments. In contrast, the assigned transitive orientation of the IG and the allocation found by our algorithm are shown in Figures 7(f) and 7(e). The gap between the two allocations is 24 (30%). So there is a need to look for an optimal solution efficiently in practice.

As described in Section 2.2, there exists one-to-one correspondence between finding an interval coloring and finding an acyclic orientation for a weighted graph. For example, the acyclic orientation α corresponding to the allocation found by First-Fit-1 in Figure 6(c) is shown in Figure 6(d). It is not transitive since (S_5, S_6), $(S_6, S_2) \in \mathcal{G}_\alpha$, but $(S_5, S_2) \notin \mathcal{G}_\alpha$. Similarly, the acyclic orientation α shown in Figure 7(d) is also not transitive since $(S_4, S_3), (S_3, S_2) \in \mathcal{G}_\alpha$ but $(S_4, S_2) \notin \mathcal{G}_\alpha$. However, the acyclic orientations β in Figure 6(f) and Figure 7(f) corresponding to the optimal allocations achieved by our algorithm are transitive. Therefore, the IGs of the programs shown in Figure 6(a) and Figure 7(a) are both comparability graph. Figure 6 and Figure 7 also illustrate the relationship between the chromatic number and the heaviest path with respect to an acyclic orientation. In Figure 6(d), the heaviest path is $S_3 \rightarrow S_5 \rightarrow S_6 \rightarrow S_2$ with a total weight of $\chi_\alpha(\mathcal{G}; w) = 208$. In Figure 6(f), the heaviest path is $S_6 \rightarrow S_2 \rightarrow S_3$ with a total weight of $\chi_\beta(\mathcal{G}; w) = 176$. In Figure 7(d), the heaviest path is $S_3 \rightarrow S_4$ with a total weight of 56. In Figure 7(f), the heaviest path is $S_4 \rightarrow S_3 \rightarrow S_2 \rightarrow S_1$ with a total weight of 80.

Our IG-based approach assumes that streams are live throughout the entire execution of any kernel that operates on them, and it is flexible enough to accommodate pre-pass optimizations applied earlier to a program by the compiler such as live range coalescing [Murthy and Bhattacharyya 2004] and live range splitting [Das et al. 2006].

4. COMPARABILITY GRAPH COLORING SRF ALLOCATION

Section 4.1 describes our key insight drawn from a careful analysis of the structure of stream IGs: a large number of stream IGs are comparability graphs, enabling their optimal colorings to be found in polynomial time. In Section 4.2, we turn

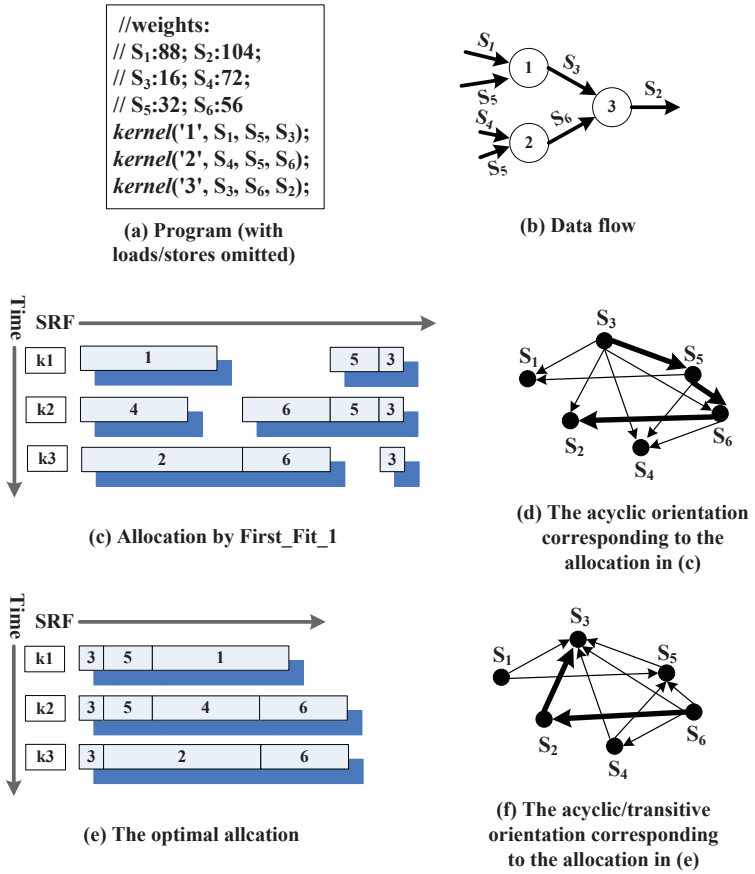


Fig. 6: An example demonstrating the superiority of our algorithm over First-Fit-1.

this insight into a procedure that can find optimal or near-optimal colorings for a well-structured media and scientific application when its stream IG is decomposable into a set of comparability graphs plus a special subgraph. There are two cases, depending on the structure of this subgraph. In Section 4.2.1, we consider the case when the subgraph is a forest, which is trivially a comparability graph. In Section 4.2.2, we consider the general case when the subgraph is an arbitrary graph, which will be completed into a comparability gragh.

4.1 Optimal Colorings of Comparability Stream IGs

In stream programs with producer-consumer locality but little global data reuse, the live ranges of streams are also local. A typical loop in such a program consists of a series of kernels, each producing intermediate streams to be consumed by the next kernel in sequence. We show below that if a stream program can be characterized as a pipeline in which each stream produced is consumed by the next actor in the pipeline. Formally, all stream live ranges in a stream IG do not span across more than two kernel calls, then the IG is a comparability graph and its optimal

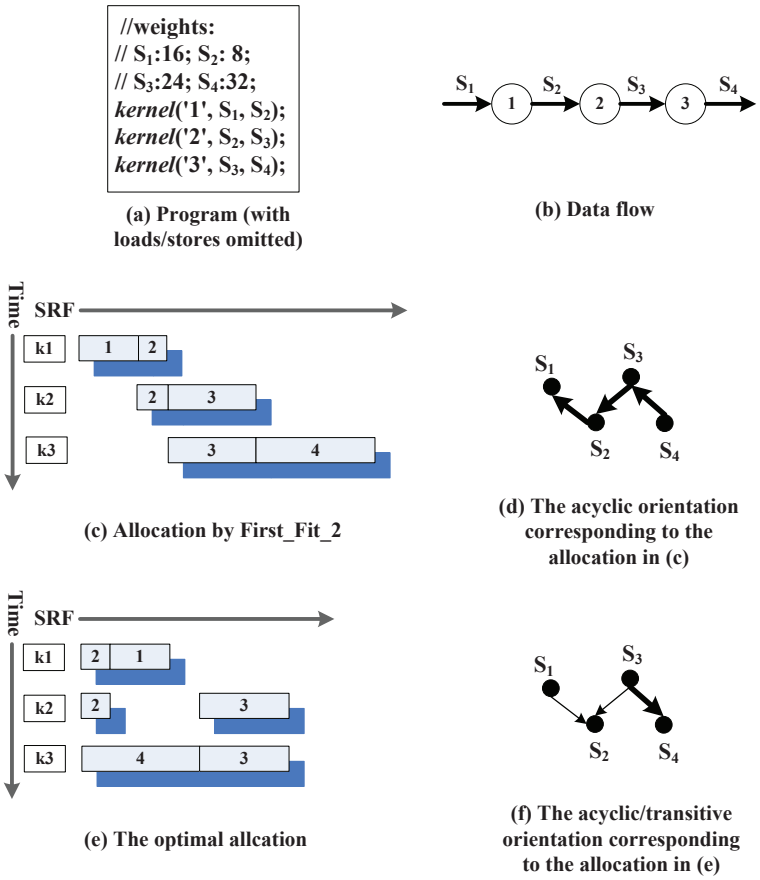


Fig. 7: An example demonstrating the superiority of our algorithm over First-Fit-2.

coloring can thus be found in polynomial time. This result is proved easily by a straightforward application of Theorem 2.5.

Figure 8 shows the IG for a series of three kernels, where no live range is longer than two kernel calls. In particular, q is live from kernel 1 to kernel 2, u, v and w are live in kernels 2 and 3, and the remaining streams are only live at the kernels where they are operated on. In this example and the proofs of our results, whether a stream is an input or output is irrelevant.

Let $\mathcal{G}_{\text{cg}}$ be the IG built from a loop containing N_{cg} kernels (numbered from 1) such that each live range in $\mathcal{G}_{\text{cg}}$ is not longer than two kernels. We partition all live ranges in $\mathcal{G}_{\text{cg}}$ into the following $2N_{\text{cg}}$ sets:

$$K_1, K_{12}, K_2, K_{23}, K_3, \ldots, K_{(N_{\text{cg}}-1)N_{\text{cg}}}, K_{N_{\text{cg}}}, K_{N_{\text{cg}}1} \tag{4}$$

where K_i consists of all streams accessed, i.e., live only in kernel i, and $K_{i(i\oplus 1)}$ all streams live only in kernels i and $i\oplus 1$; Here we define $i\oplus c$ to be $(i+c-1)\%N_{\text{cg}}+1$ and $i\ominus c$ to be $(i-c-1)\%N_{\text{cg}}+1$.

As shown in Figure 9, all streams accessed in a kernel invoked in a loop form a

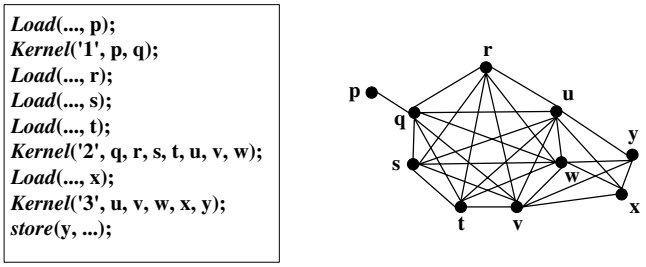


Fig. 8: A stream program and its IG.

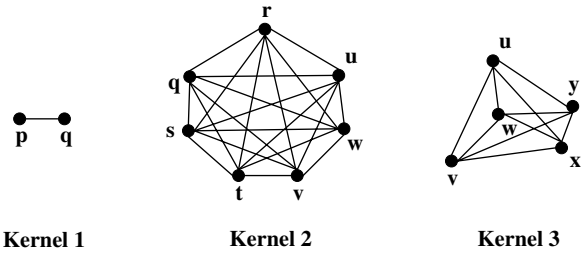


Fig. 9: Kernel-induced cliques for Figure 8.

maximal clique in the stream IG of the loop. Furthermore, the following result is obvious.

LEMMA 4.1. *The streams in* $K_{(i\ominus 1)i} \cup K_i \cup K_{i(i\oplus 1)}$ *form a maximal clique for every kernel* i.

Our main results are stated in two theorems, Theorem 4.2 applies when $N_{\rm cg}$ is even and Theorem 4.3 applies when $K_{N_{\rm cg}1} = \emptyset$, i.e., when cross-iteration reuse is absent. If neither condition holds, we can apply loop unrolling once to produce a loop with an even number of kernels so that Theorem 4.2 can be used. For stream processors, unrolling a loop that is executed on the host does not affect negatively program performance (since code size expansion for the host is not a concern).

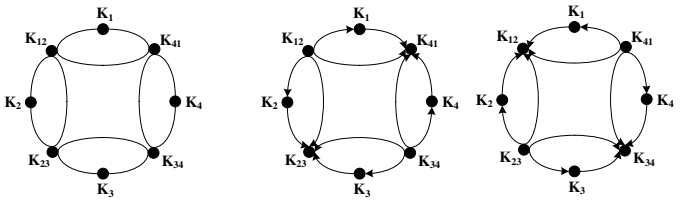
THEOREM 4.2. *If* $N_{\rm cg}$ *is even,* $\mathcal{G}_{\rm cg}$ *is a comparability graph.*

PROOF. Let us assume first that all sets listed in (4) are not empty. By construction, the live ranges in every such a set are equal. Thus, the induced subgraph of $\mathcal{G}_{\rm cg}$ by K_i ($K_{i(i\oplus 1)}$) is a clique, denoted $\mathcal{G}_i$ ($\mathcal{G}_{i(i\oplus 1)}$). So we have the following $2N_{\rm cg}$ induced cliques:

$$\mathcal{G}_1, \mathcal{G}_{12}, \mathcal{G}_2, \mathcal{G}_{23}, \mathcal{G}_3, \ldots, \mathcal{G}_{(N_{\rm cg}-1)N_{\rm cg}}, \mathcal{G}_{N_{\rm cg}}, \mathcal{G}_{N_{\rm cg}1} \tag{5}$$

In addition, for any two sets K and K' listed in (4), either every live range $x \in K$ interferes with every live range $x' \in K'$ or there is no interference between the live ranges in K and those in K'.

By Theorem 2.5, in $\mathcal{G}_{\rm cg}$, if we let $\mathcal{G}_i$ ($\mathcal{G}_{i(i\oplus 1)}$) "collapse" into one node, identified



(a) $\mathcal{G}_0$ (b) Two orientations

Fig. 10: Two transitive orientations of $\mathcal{G}_0$ ($N_{\rm cg} = 4$).

by K_i ($K_{i(i\oplus 1)}$), and denote the resulting "decomposed graph" by $\mathcal{G}_0$, we have:

$$\mathcal{G}_{\rm cg} = \mathcal{G}_0[\mathcal{G}_1, G_{12}, G_2, \ldots, \mathcal{G}_{N_{\rm cg}}, \mathcal{G}_{N_{\rm cg}1}]$$

A clique is a comparability graph. Thus, all $\mathcal{G}_i$'s given in (5) are comparability graphs. Then, by Theorem 2.5, $\mathcal{G}_{\rm cg}$ is a comparability graph if we show that $\mathcal{G}_0$ is. To achieve this, by Definition 2.2, it suffices if we can find a transitive orientation of $\mathcal{G}_0$. As shown in Figure 10, there are exactly two different transitive orientations since $K_{12}, K_{23}, \ldots, K_{N_{\rm cg}1}$ must alternate to be a source or a sink. To see this, consider $K_{i(i\oplus 1)}$, which is adjacent to $K_{(i\ominus 1)i}, K_i, K_{i\oplus 1}$ and $K_{(i\oplus 1)(i\oplus 2)}$. Suppose the orientation assigned to edge $(K_i, K_{i(i\oplus 1)})$ is $K_i \rightarrow K_{i(i\oplus 1)}$. Since K_i is not adjacent to $K_{i\oplus 1}$, the orientation of edge $(K_{i(i\oplus 1)}, K_{i\oplus 1})$ is forced to be $K_{i\oplus 1} \rightarrow K_{i(i\oplus 1)}$. Otherwise, $\mathcal{G}_0$ cannot be transitive. Similarly, the orientation of $(K_{i(i\oplus 1)}, K_{(i\oplus 1)(i\oplus 2)})$ must be $K_{(i\oplus 1)(i\oplus 2)} \rightarrow K_{i(i\oplus 1)}$. Since the orientation of $(K_{i(i\oplus 1)}, K_{i\oplus 1})$ is $K_{i\oplus 1} \rightarrow K_{i(i\oplus 1)}$, the orientation of $(K_{(i\ominus 1)i}, K_{i(i\oplus 1)})$ can only be $K_{(i\ominus 1)i} \rightarrow K_{i(i\oplus 1)}$ as $K_{(i\ominus 1)i}$ is not adjacent to $K_{i\oplus 1}$. Therefore, once the orientation of $(K_i, K_{i(i\oplus 1)})$ is assigned, the orientations for all the other incident edges of $K_{i(i\oplus 1)}$ are identically assigned, making $K_{i(i\oplus 1)}$ either a source or a sink. Inductively, $K_{12}, K_{23}, \ldots, K_{N_{\rm cg}1}$ must alternate to be a source or a sink. This is possible since $N_{\rm cg}$ is even.

Finally, if any set listed in (4) is empty, $\mathcal{G}_0$ is still a comparability graph since every induced subgraph of a comparability graph is a comparability graph. □

THEOREM 4.3. *If $K_{N_{\rm cg}1} = \emptyset$, $\mathcal{G}_{\rm cg}$ is a comparability graph.*

PROOF. A transitive orientation of $\mathcal{G}_{\rm cg}$ always exists since the"ring" as shown in Figure 10 is broken at $K_{N_{\rm cg}1}$. □

In fact, Theorem 4.3 holds whenever $K_{i(i\oplus 1)} = \emptyset$ for some i.

Let us illustrate Theorem 4.3 in Figure 11 for the IG shown in Figure 8. Being a comparability graph, its optimal coloring is guaranteed. The optimality is independent of the node weights in the graph.

The facts stated in Lemmas 4.4 and 4.5 given below are exploited in the development of our algorithm for coloring stream IGs in Section 4.2.1.

LEMMA 4.4. *Suppose $\mathcal{G}_{\rm cg}$ is a comparability graph. Let $\mathcal{G}'_{\rm cg}$ be an induced subgraph of $\mathcal{G}_{\rm cg}$. If $\mathcal{G}'_{\rm cg}$ is connected, then it has at most eight different transitive orientations.*

PROOF. If $\mathcal{G}'_{\rm cg}$ is connected, there must exist two kernels i and j such that $K_{i(i\oplus 1)}, K_{(i\oplus 1)(i\oplus 2)}, \ldots K_{(j\ominus 2)(j\ominus 1)}, K_{(j\ominus 1)j}$ are in $\mathcal{G}'_{\rm cg}$. and that these are the only

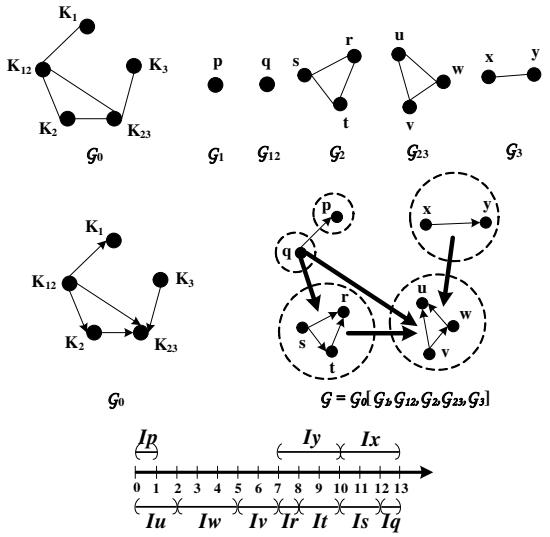


Fig. 11: Optimal interval coloring of the stream IG given in Figure 8 (with the weights of p, q and r being 1, the weights of s, t, u and v being 2 and the weights of w, x and y being 3). To avoid cluttering, in the graph labelled by $\mathcal{G}_0[\mathcal{G}_1, \mathcal{G}_{12}, \mathcal{G}_2, \mathcal{G}_{23}, \mathcal{G}_3]$, a thicker arrow directing from a clique K to a clique K' symbolizes all directed edges (x, y), for all $x \in K$ and all $y \in K'$.

sets containing two-kernel long live ranges listed in (4) in $\mathcal{G}'_{\text{cg}}$. Due to space limitation, we do not enumerate all the cases. Instead, we discuss only the case with the largest number of transitive orientations. In this case, $K_i = K_j = \emptyset$, $j - i > 2$. In a transitive orientation of $\mathcal{G}'_{\text{cg}}$, the middle $j - i - 2$ sets in the above list must alternate to serve as a source or a sink. So there are only two possibilities. In either case, edge $(K_{i(i\oplus 1)}, K_{i\oplus 1})$ may have at most two orientations, and similarly, edge $(K_{j\ominus 1}, K_{(j\ominus 1)j})$ may have at most two orientations. So there are at most $2 \times 2 \times 2 = 8$ different transitive orientations. □

LEMMA 4.5. *Let $\mathcal{G}_{\text{cg}}$ be a comparability graph. If all sets in (4) are nonempty, $\mathcal{G}_{\text{cg}}$ has two transitive orientations.*

Proof. $\mathcal{G}_{\text{cg}}$ is connected and then apply Lemma 4.4. □

4.2 A General Algorithm for Coloring Stream IGs

In some scientific applications (amenable to stream processing), the presence of temporal reuse in a few streams could make their live ranges longer than two kernels. In some media applications, there are also occasionally a few long producer-consumer live ranges. Furthermore, some live ranges may be extended by the programmer or a pre-pass compiler optimization in order to overlap memory transfers and kernel execution. Such stream IGs may or may not be comparability graphs. In this section, we generalize our work described in the preceding section to deal with these stream IGs, resulting in an SRF allocation algorithm, CGC, given in Algorithm 1.

The basic idea is to partition the node set V in $\mathcal{G} = (V, E)$ into the following two

Algorithm 1 Coloring an arbitrary stream IG.

```
1: procedure CGC
2: Input: G = (V, E) with V = {V_s, V_l} and E = {E_s, E_l, E_sl}
3: Output: An acyclic orientation α of G
4: if a transitive orientation α of G can be found then
5:    return α
6: end if
7: if G(V_l) is a forest then
8:    α = FOREST_CGC(G)
9: else
10:   α = GEN_CGC(G)
11: end if
12: return α
13: end procedure
```

subsets:

$$V_s = \{v \in V \mid v\text{'s live range spans at most 2 kernels}\}$$
$$V_l = \{v \in V \mid v\text{'s live range spans more than 2 kernels}\}$$

Thus, E is partitioned into the following three subsets:

$$E_s = \{(x,y) \in E \mid x \in V_s, y \in V_s\}$$
$$E_l = \{(x,y) \in E \mid x \in V_l, y \in V_l\}$$
$$E_{sl} = \{(x,y) \in E \mid x \in V_s, y \in V_l\}$$

By Theorems 4.2 and 4.3, the subgraph $\mathcal{G}(V_s)$ induced by V_s is a comparability graph. Our key observation, from both the characteristics of stream applications and the codes of the benchmarks (Section 5), is that the long live ranges in stream IGs are sparse and tend not to be live simultaneously. As a result, in most cases, the subgraph $\mathcal{G}(V_l)$ is a forest of disjoint trees.

As shown in Algorithm 1, if $\mathcal{G}$ is a comparability graph (Definition 2.2), then by Theorem 2.3, an optimal coloring, represented by a transitive orientation, is returned immediately (lines 4 – 6). Otherwise, CGC works by distinguishing the two cases depending on if $\mathcal{G}(V_l)$ is a forest or not, which are discussed separately in Sections 4.2.1 and 4.2.2.

4.2.1 $\mathcal{G}(V_l)$ *Is a Forest.* As discussed earlier, the long live ranges in stream applications, i.e., in $\mathcal{G}(V_l)$ tend to form a forest of disjoint trees. We discuss below how FOREST_CGC given in Algorithm 2 handles such commonly occurring cases in practice.

As illustrated in Figure 12, V_l consists of two long live ranges m and p: m is live from kernel 2 to kernel 5 and p is live from kernel 3 to kernel 6. Since both streams interfere with each other, the forest $\mathcal{G}(V_l)$ has only one tree, which is a line connecting m and p.

THEOREM 4.6. *A forest is a comparability graph. Let the number of trees in the forest* $\mathcal{G}(V_l)$ *be* $\mathsf{trees}(\mathcal{G}(V_l))$. *Then* $\mathcal{G}(V_l)$ *has a total of* $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ *transitive orientations.*

Proof. A forest consists of disjoint trees. Thus, a forest is a comparability graph

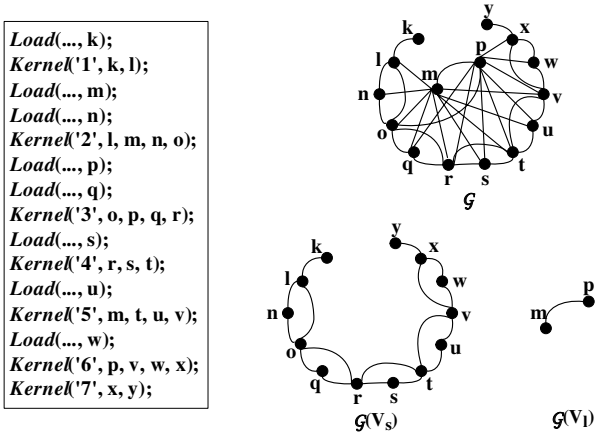


Fig. 12: A program with two long live ranges m and p.

if and only if a tree is. A tree $\mathcal{G} = (V, E)$ is bipartite [West 1996]. So V can be partitioned into two disjoint sets $V = S_1 + S_2$ such that every edge has one endpoint in S_1 and the other in S_2. It is easy to obtain two transitive orientations of $\mathcal{G}$ by orienting all the edges from S_1 to S_2, and vice versa, as shown in Figure 13. Thus, a tree is a comparability graph. So does a forest.

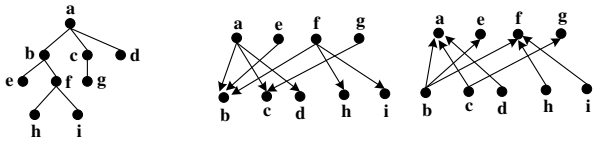


Fig. 13: Two transitive orientations of a tree.

A tree with more than one node has exactly two transitive orientations [West 1996], as shown in Figure 13. Thus, $\mathcal{G}(V_l)$ has a total of $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ transitive orientations. □

In Section 4.2.1.1, we describe the algorithm behind FOREST_CGC for coloring commonly occurring stream IGs. In Section 4.2.1.2, we argue that why it tends to give optimal and near-optimal colorings in practice.

4.2.1.1 *Algorithm.* As shown in Algorithm 2, if $\mathcal{G}$ is not a comparability graph and $\mathcal{G}(V_l)$ is a forest, in which case, an optimal or near-optimal coloring, represented by an acyclic orientation, is found in three steps, motivated by (3). Recall that $\mathsf{trees}(\mathcal{G}(V_l))$ be the number of trees in $\mathcal{G}(V_l)$. The basic idea is to enumerate the set $\mathcal{O}_s$ of all transitive orientations of E_s, i.e., $G(V_s)$ (in Step 1) and enumerate the set $\mathcal{O}_l$ of all $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ transitive orientations for the forest E_l, i.e., $\mathcal{G}(V_l)$ (in Step 2). As a result, for every possible combination $o_s \times o_l$ in $\mathcal{O}_s \times \mathcal{O}_l$, a unique orientation to E_{sl} is determined (in Step 3). Among $|\mathcal{O}_s| \times |\mathcal{O}_l|$ acyclic orientations of $\mathcal{G}$ found, the one whose heaviest (directed) path is the smallest is returned (lines 12 – 14).

Algorithm 2 Coloring when $\mathcal{G}(V_l)$ is a forest.

1: **procedure** FOREST_CGC
2: **Input:** $\mathcal{G} = (V, E)$ with $V = \{V_s, V_l\}$ and $E = \{E_s, E_l, E_{sl}\}$
3: **Output:** An acyclic orientation α of $\mathcal{G}$
4: $\chi_{\min}(\mathcal{G}) = +\infty$
5: Let $\mathcal{O}_s$ be the set of all transitive orientations of E_s, i.e., $\mathcal{G}(V_s)$
6: Let $\mathcal{O}_l$ be the set of $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ transitive orientations of E_l, i.e., $\mathcal{G}(V_l)$
7: **for** each orientation $o_s \times o_l \in \mathcal{O}_s \times \mathcal{O}_l$ **do**
8: **for** $(x, y) \in E_{sl}$, where $x \in V_s$ and $y \in V_l$ **do**
9: Direct an arc from y to x (x to y) if y is a source (sink)
10: **end for**
11: Let α be the acyclic orientation of $\mathcal{G}$ thus found
12: **if** $\chi_\alpha(\mathcal{G}) < \chi_{\min}(\mathcal{G})$ **then**
13: $\chi_{\min}(\mathcal{G}) = \chi_\alpha(\mathcal{G})$
14: Record the α as the current best
15: **end if**
16: **end for**
17: **return** α
18: **end procedure**

We consider only the transitive orientations in $\mathcal{G}(V_s)$ and $\mathcal{G}(V_l)$ in order to reduce the underlying solution space and the width of the final interval coloring found.

In Step 1 (line 5), the set $\mathcal{O}_s$ of all transitive orientations of E_s, i.e., $\mathcal{G}(V_s)$ is found. In real code (the benchmarks described in Section 5), $\mathcal{G}(V_s)$ is generally connected, resulting in exactly two transitive orientations by Lemma 4.5 as illustrated in Figure 10. There can be only a limited number of transitive orientations when $\mathcal{G}(V_s)$ is disconnected since the number of transitive orientations of each connected subgraph is bounded by Lemma 4.4.

In Step 2 (line 6), we find all $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ transitive orientations of the trees in $\mathcal{G}(V_l)$, i.e., E_l.

In Step 3 (lines 7 – 10), for each orientation $o_s \times o_l \in \mathcal{O}_s \times \mathcal{O}_l$ (line 7), a unique orientation of E_{sl} is fixed (lines 8 – 10). For each edge $(x, y) \in E_{sl}$, where $x \in V_s$ and $y \in V_l$, its orientation is assigned based on the property of y. From Figure 13, it can be easily observed that y is either a source or a sink under o_l. If y is a source, direct the edge from y to x, namely, maintain y's property; otherwise, direct the edge from x to y.

Every orientation α of $\mathcal{G}$ found in line 11 is acyclic. This can be reasoned about as follows. No directed path confined to $\mathcal{G}(V_s)$ can be a cycle since $\mathcal{G}(V_s)$ is a comparability graph. In addition, no directed path that contains a node in $\mathcal{G}(V_l)$ can be a cycle since the node must be either a source or a sink (Figure 13).

FOREST_CGC is polynomial in practice. For comparability graphs, their recognition and optimal colorings are polynomial. In addition, $\mathcal{G}(V_s)$ is mostly connected, resulting in a few orientations (Lemmas 4.4 and 4.5). Finally, $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ is small since $\mathcal{G}(V_l)$ has a few trees.

Let us apply FOREST_CGC to the program given in Figure 12. In lines 4 – 6 in Algorithm 1, $\mathcal{G}$ is detected to be a comparability graph. Its optimal coloring is found and returned immediately. Nevertheless, let us use this example to explain how

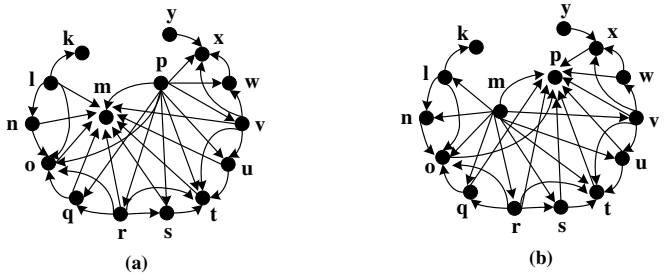


Fig. 14: Two orientations for the program in Figure 12.

FOREST_CGC works. $\mathcal{G}(V_s)$ is connected and happens to have only two transitive orientations. As $\mathcal{G}(V_l)$ has only one tree, there are two orientations. So there are a total of four orientations, two of which are shown in Figure 14. The one in Figure 14(a) will be the solution found since it is transitive.

4.2.1.2 *Analysis.* We now argue that FOREST_CGC finds optimal colorings for most stream programs. We show further that non-optimal colorings occur only infrequently and are near-optimal in the sense that they are only larger by the sum of one or two stream sizes in the worst case. Our claim is validated in our experiments in Section 5.

Recall that $\chi(\mathcal{G}; w)$ denotes the chromatic number of $\mathcal{G}$. In FOREST_CGC, α is the best acyclic orientation found and $\chi_\alpha(\mathcal{G}; w)$ is its width. Let P_α be the heaviest directed path in $\mathcal{G}_\alpha$ of the following form:

$$P_\alpha =_{\text{def}} v_1, v_2, \ldots, v_m \tag{6}$$

According to (3), we have $\chi_\alpha(\mathcal{G}; w) = w(P_\alpha)$. In addition, $w(P_\alpha)$ is the smallest among the heaviest directed paths in all $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ orientations of $\mathcal{G}$ found in line 14 of FOREST_CGC given in Algorithm 2.

Definition 4.7. FOREST_CGC is *optimal* for $\mathcal{G}$ if $\chi_\alpha(\mathcal{G}; w) = \chi(\mathcal{G}; w)$.

All results presented below in this section are formulated and proved based on reasoning about the structure of P_α, which is uncovered in Lemma 4.8, resulting in four cases to be distinguished, and Lemma 4.9. In two cases, FOREST_CGC is optimal (Theorem 4.10). In the remaining two cases, FOREST_CGC is also optimal for many stream IGs. Non-optimal solutions α are returned only infrequently when some strict conditions are met, and moreover, these solutions are near-optimal since $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w)$ is small for reasonably large stream IGs (Theorems 4.11 and 4.12).

LEMMA 4.8. *Only v_1 or v_m may appear in $\mathcal{G}(V_l)$.*

PROOF. Follows from the fact that for every orientation α of $\mathcal{G}$ found in line 11 of FOREST_CGC, every node in $\mathcal{G}(V_l)$ is either a source or a sink under α. □

This lemma implies that all nodes in P_α are contained in $\mathcal{G}(V_s)$ except its start and end nodes.

Let $\mathcal{K}_i = K_{(i\ominus 1)i} \cup K_i \cup K_{i(i\oplus 1)}$. By Lemma 4.1, $\mathcal{K}_i$ is a maximal clique in $\mathcal{G}(V_s)$ (as illustrated in Figure 9).

LEMMA 4.9. *The nodes of P_α contained in $\mathcal{G}(V_s)$ form a clique $\mathcal{K}_i$ for some i, where $\mathcal{K}_i = K_{(i\ominus 1)i} \cup K_i \cup K_{i(i\oplus 1)}$.*

PROOF. α found by FOREST_CGC is a transitive orientation of $\mathcal{G}(V_s)$. Without loss of generality, suppose the nodes of P_α in $\mathcal{G}(V_s)$ are $v_2, v_3, \ldots, v_{m-1}$.

First, there cannot exist two different nodes v_i and v_j in $v_2, v_3, \ldots, v_{m-1}$, where $i < j$, such that $v_i \in K_i$ and $v_j \in K_j$. Otherwise, since $\{v_i, \ldots, v_j\} \subseteq P_\alpha$ is a directed path in $\mathcal{G}_\alpha$ and α is transitive, then $(v_i, v_j) \in \mathcal{G}(V_s)$. However, $(v_i, v_j) \notin \mathcal{G}(V_s)$ because v_i and v_j do not interfere with each other. A contradiction.

Second, there cannot exist three distinct nodes v_i, v_j and v_k in $v_2, v_3, \ldots, v_{m-1}$, where $i < j < k$, such that $v_i \in K_{i(i\oplus 1)}$, $v_j \in K_{j(j\oplus 1)}$ and $v_k \in K_{k(k\oplus 1)}$. Otherwise, since $\{v_i, \ldots, v_j, \ldots, v_k\} \subseteq P_\alpha$ is a directed path in $\mathcal{G}_\alpha$ and α is transitive, then $(v_i, v_k) \in \mathcal{G}(V_s)$. However, $(v_i, v_k) \notin \mathcal{G}(V_s)$ due to v_i and v_k do not interfere with each other (the only exception is when $N_{cg} = 3$, and $K_{31} \neq \emptyset$, however, in that case, according to Theorem 4.2 and Theorem 4.3, the loop will be unrolled once beforehand). A contradiction.

Third, it is not possible for $v_2, v_3, \ldots, v_{m-1}$ to be contained in two "non-consecutive" $K_{(i\ominus 1)i}$ and $K_{j(j\oplus 1)}$, where $j \neq i$ and $i \ominus 1 \neq j \oplus 1$. Otherwise, we would end up in a situation that contradicts to the fact just established (in the second step).

So $v_2, v_3, \ldots, v_{m-1}$ are all contained in $\mathcal{K}_i$ for some i.

Finally, $\mathcal{K}_i$ is a clique, which must be formed by $v_2, v_3, \ldots, v_{m-1}$ since P_α is the heaviest path found by α. □

There are four cases depending on the structure of P_α:

—Case P1. P_α is contained in $\mathcal{G}(V_s)$
—Case P2. P_α is contained in $\mathcal{G}(V_l)$
—Case P3. v_1, v_m are both contained in $\mathcal{G}(V_l)$
—Case P4. Either v_1 or v_m is in $\mathcal{G}(V_l)$ (but not both)

THEOREM 4.10. FOREST_CGC *is optimal in Cases P1 and P2.*

PROOF. In Case P1, $\mathcal{G}(V_s)$ is a comparability graph. Thus, P_α must be contained in a clique K in $\mathcal{G}(V_s)$ (and also in $\mathcal{G}$). This means that $\chi_\alpha(\mathcal{G}; w) = w(P_\alpha) = w(K)$. Since $w(K) \leqslant \chi(\mathcal{G}; w)$, we must have $\chi_\alpha(\mathcal{G}; w) \leqslant \chi(\mathcal{G}; w)$. So α is optimal. The proof for Case P2 is similar since the forest $\mathcal{G}(V_l)$ is also a comparability graph. □

THEOREM 4.11. *In Case P3, we have $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) \leqslant w(v_1) + w(v_m)$, where the equality holds if and only if $v_2, v_3, \ldots v_{m-1}$, happen to form the heaviest clique $\mathcal{K}$ in $\mathcal{G}$ such that $\chi(\mathcal{G}; w) = w(\mathcal{K})$.*

PROOF. By Lemma 4.8 and Lemma 4.9, we have $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) \leqslant w(v_1) + w(v_m)$. We now prove the "if" and "only if" for the equality. The "if" part is true since $\chi_\alpha(\mathcal{G}; w) = w(P_\alpha) = w(v_1) + w(v_m) + w(\mathcal{K}) = w(v_1) + w(v_m) + \chi(\mathcal{G}; w)$. The "only if" part is true due to Lemma 4.9 and the given hypothesis $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) = w(v_1) + w(v_m)$. □

An analogue of Theorem 4.11 for Case P4 is given below.

THEOREM 4.12. *In Case P4, suppose that v_1 is contained in $\mathcal{G}(v_l)$ but v_m is not. Then $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) \leqslant w(v_1)$, where the equality holds if and only if $v_2, v_3, \ldots v_m$, happens to form the heaviest clique $\mathcal{K}$ in $\mathcal{G}$ such that $\chi(\mathcal{G}; w) = w(\mathcal{K})$.*

Algorithm 3 Coloring when $\mathcal{G}(V_l)$ is not a forest.

1: **procedure** GEN_CGC
2: **Input:** $\mathcal{G} = (V, E)$ with $V = \{V_s, V_l\}$ and $E = \{E_s, E_l, E_{sl}\}$
3: **Output:** An acyclic orientation α of $\mathcal{G}$
4: $\mathcal{G}' = (V, E') =$ **Find_Max_Forest_Subgraph**$(\mathcal{G})$
5: $\alpha' =$ FOREST_CGC $(\mathcal{G}')$
6: Let $E_m = E - E'$, where $E_m \subset E_l$
7: $\chi_{\min}(\mathcal{G}) = +\infty$
8: Let $\mathcal{O}_m$ be the set of $2^{|E_m|}$ orientations of E_m
9: **for** each orientation $o_m \in \mathcal{O}_m$ **do**
10: Let $\alpha = \alpha' \cup o_m$ be an orientation of $\mathcal{G}$
11: **if** α is acyclic **then**
12: **if** $\chi_\alpha(\mathcal{G}) < \chi_{\min}(\mathcal{G})$ **then**
13: $\chi_{\min}(\mathcal{G}) = \chi_\alpha(\mathcal{G})$
14: Record the α as the current best
15: **end if**
16: **end if**
17: **end for**
18: **Output:** α
19: **end procedure**
20: **procedure** Find_Max_Forest_Subgraph
21: **Input:** $\mathcal{G} = (V, E)$ with $V = \{V_s, V_l\}$ and $E = \{E_s, E_l, E_{sl}\}$
22: **Output:** a subgraph $\mathcal{G}'$ of $\mathcal{G}$, such that $\mathcal{G}'(V_l)$ is a forest
23: Let $\mathcal{G}_{lf} = (V_l, E_l')$ be a maximum spanning forest of $\mathcal{G}(V_l)$
24: Let $E' = E_s + E_{sl} + E_l'$
25: Let $\mathcal{G}'$ be the subgraph of $\mathcal{G}$ formed by V and E'
26: **return** $\mathcal{G}'$
27: **end procedure**

4.2.2 *$\mathcal{G}(V_l)$ Is Not a Forest.* In the rare cases when $\mathcal{G}(V_l)$ is not a forest, GEN_CGC given in Algorithm 3 first finds a subgraph $\mathcal{G}'$ of $\mathcal{G}$, such that $\mathcal{G}'(V_l)$ is a maximum spanning forest of $\mathcal{G}(V_l)$, and then invokes FOREST_CGC to obtain the optimal orientation α' of $\mathcal{G}'$ (lines 4 – 5 in Step 1). Then GEN_CGC enumerates the set $\mathcal{O}_m$ of all $2^{|E_m|}$ orientations for the edge set $E_m \subset E_l$ (lines 6 – 8 in Step 2). As a result, for every $o_m \in \mathcal{O}_m$, combined with α', an orientation of $\mathcal{G}$ is determined. Among $2^{|E_m|}$ orientations of $\mathcal{G}$ found, the acyclic one whose heaviest (directed) path is the smallest is returned (lines 9 – 18 in Step 3).

GEN_CGC is polynomial in practice. Find_Max_Forest_Subgraph can be done in $O(\mid V_l \mid + \mid E_l \mid)$ time. An orientation can be determined to see if it is acyclic or not in $O(\mid V \mid + \mid E \mid)$ time by the algorithm for topological sorting [Kahn 1962]. Finally, $2^{|E_m|}$ is small since $\mid E_l \mid$ is small because $\mid V_l \mid$ is very small, and $E_m \subset E_l$. According to our experiments described in Section 5, $\mid E_m \mid$ is mostly smaller than 10, with a maximum value of 13.

Figure 15(a) shows a stream program with a series of six kernels. Its IG is depicted in Figure 15(b). V_l consists of three long live ranges c, e and f: c is live from kernel 1 to kernel 5, e is live from kernel 2 to kernel 6, and f is live from kernel 2 to kernel 5. Since all streams interfere with each other, the subgraph $\mathcal{G}(V_l)$ induced by V_l is a clique as shown in Figure 15(d) but not a forest. Let us apply

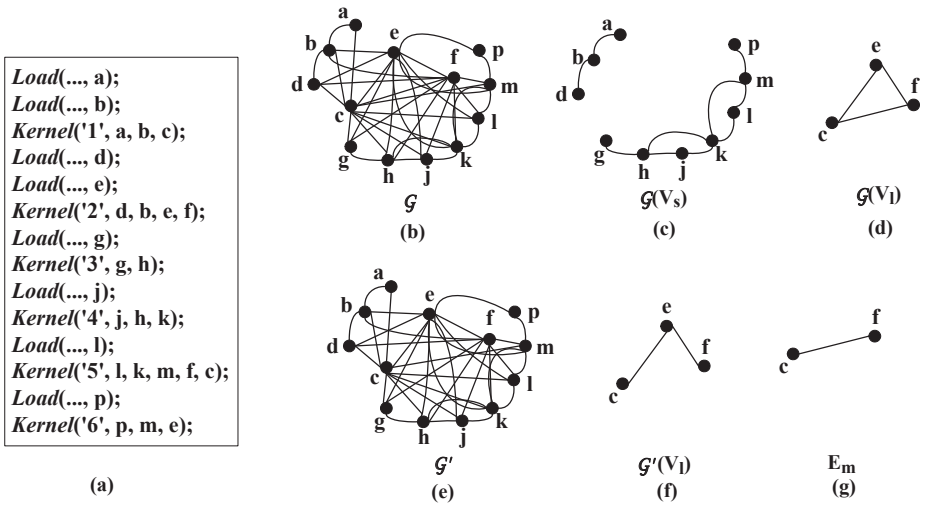


Fig. 15: A program with three long live ranges c, e and f, and $\mathcal{G}(V_l)$ is not a forest.

GEN_CGC to the IG given in Figure 15(b). In line 4, the subgraph $\mathcal{G}'$ of $\mathcal{G}$ computed is depicted in Figure 15(e). There is only one edge (c, f) absent: $\mathcal{G}'(V_l)$ is shown in Figure 15(f) and E_m is shown in Figure 15(g). In line 5, the optimal orientation α' of $\mathcal{G}'$ is computed by FOREST_CGC. Next, in lines 8 – 18, the set $\mathcal{O}_m$ of all $2^{|E_m|} = 2$ orientations for E_m is enumerated. Combined with α', the $2^{|E_m|} = 2$ orientations of $\mathcal{G}$ are induced, and from which the optimal one is returned.

5. EXPERIMENTS

Research into advanced compiler technology for stream processing is still at its infancy. There are presently no standard benchmarks available. Table I gives a list of 13 media and scientific applications available to us for the FT64 stream processor. NLAG-5 is a nonlinear algebra solver for two-dimensional nonlinear diffusion of hydrodynamic. QMR is the core iteration in the QMRCGSTAB algorithm for solving nonsymmetric linear systems. LUD is a dense LU Decomposition solver. Viterbi implements the Viterbi decoding algorithm [Viterbi 1967]. Triangle Rendering is referred to [Rixner et al. 1998]. As shown, the stream IGs in 12 benchmarks are comparability graphs. Their optimal colorings are guaranteed. The $\mathcal{G}(V_l)$ for QMR is a forest as depicted in Figure 16, it is a comparability graph. A transitive orientation of this comparability graph is shown in Figure 17. For small applications (the real benchmarks shown above we currently have), either First-Fit or our algorithm suffices. For large applications, First-Fit can be sub-optimal, as validated below.

In Section 5.1, we demonstrate that CGC can find optimal and nearly optimal colorings efficiently for a large number of randomly generated stream IGs when $\mathcal{G}(V_l)$ is a forest. In Section 5.2, we demonstrate further the effectiveness of CGC in coloring stream IGs in the rare cases when $\mathcal{G}(V_l)$ is not a forest.

Benchmark	Source	IG
Laplace	NCSA	C
Swim-calc1	SPEC2000	C
Swim-calc2	SPEC2000	C
GEMM	BLAS	C
FFT	-	C
EP	NPB	C
NLAG-5	-	C
QMR	-	F
LUD	-	C
Jacobi	-	C
MG	NPB	C
Viterbi	-	C
Triangle Rendering	-	C

Table I: Media and scientific programs. C (F) marks a stream IG $\mathcal{G}$ ($\mathcal{G}(V_l)$) to be a comparability graph (forest).

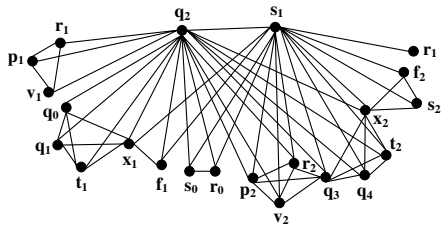


Fig. 16: The IG for the unrolled QMR.

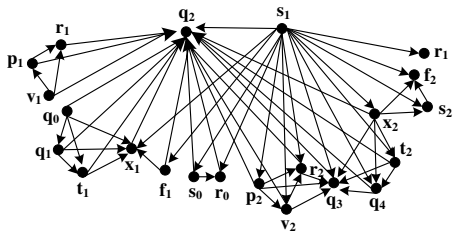


Fig. 17: A transitive orientation for (unrolled) QMR.

5.1 $\mathcal{G}(V_l)$ Is a Forest

In this case, FOREST_CGC is invoked. We have implemented an algorithm that randomly generates the stream IGs that satisfy the stream application characteristics exploited in the development of FOREST_CGC as discussed in Section 4.2.1. All random numbers are in discrete uniform distribution generated by *unidrnd* in Matlab unless specified otherwise.

There are five steps in building a stream IG $\mathcal{G}$. Step 1 generates the number of kernels, denoted *num_kernel*. In Step 2, we obtain the set of short live ranges, namely $G(V_s)$. For each kernel i, we generate two sets K_i and $K_{i(i\oplus 1)}$. We generate

No.	$G^1_{\mathcal{L}}$			$G^2_{\mathcal{U}}$			$G^3_{\mathcal{U}}$			$G^3_{\mathcal{L}}$		
	N	E	T	N	E	T	N	E	E_m	N	E	E_m
1	38	160	1	213	880	4	113	495	4	177	794	1
2	152	565	2	248	990	4	57	273	1	95	368	3
3	122	492	2	220	793	1	149	636	2	68	342	3
4	63	265	1	283	1039	1	173	770	2	87	329	3
5	111	449	1	255	1015	2	54	259	1	116	537	3
6	180	730	4	230	919	6	66	311	2	182	821	2
7	131	522	2	350	1479	4	177	794	1	137	633	2
8	142	562	1	409	1836	8	35	150	1	187	742	1
9	91	367	2	219	875	7	95	368	3	57	303	1
10	21	56	1	307	1115	4	68	342	3	34	139	1
11	192	746	3	394	1528	4	87	329	3	164	702	6
12	129	549	2	323	1342	6	116	537	3	122	491	1
13	80	378	2	273	1147	7	137	633	2	171	741	1
14	41	151	1	306	1248	4	187	742	1	73	348	1
15	44	159	1	360	1451	6	57	303	1	123	549	13
16	116	504	3	208	794	3	183	781	5	34	145	1
17	59	216	1	339	1298	2	34	149	1	59	308	1
18	97	334	2	345	1416	6	119	609	7	150	681	3
19	137	506	3	235	941	5	164	702	6	83	408	8
20	114	455	2	308	1151	3	122	491	1	136	633	2
21	54	238	2	265	1070	5	171	714	1	172	930	10
22	110	434	2	392	1577	5	73	348	1	170	828	5
23	137	498	1	301	1170	5	123	549	13	116	559	1
24	51	214	2	313	1167	4	34	145	1	214	1084	6
25	216	953	5	399	1649	9	66	311	2	105	493	1
26	166	711	5	306	1230	3	150	681	3	170	845	5
27	75	319	2	255	898	2	83	408	8	109	485	5
28	166	664	5	297	1150	3	136	633	2	159	648	1
29	132	508	2	326	1203	1	172	930	10	118	557	2
30	167	653	3	292	1111	2	170	828	5	127	589	1

Table II: Results for four groups of weighted IG graphs, where N stands for the number of nodes, E the number of edges, T the number of trees in $\mathcal{G}(V_l)$, and E_m the number of edges which are not in the spanning forest of $\mathcal{G}(V_l)$.

one random number in the range [1,3] to represent the number of live ranges in K_i and another random number in [1,3] to represent the number of live ranges in $K_{i(i\oplus 1)}$. Thus, each kernel has at most nine short live range streams live at the kernel: three from $K_{(i\ominus 1)i}$, three from K_i and three from $K_{i(i\oplus 1)}$.

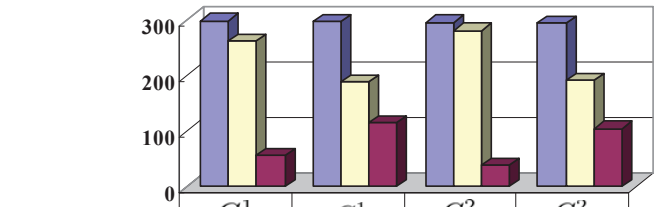
In Step 3, we generate the set of long live ranges, namely $G(V_l)$. We generate a random number p ranging from 1% to 20% to represent the percentage of long live ranges in $\mathcal{G}(V_l)$ over *num_kernel*. Thus, $|\mathcal{G}(V_l)| = p \times$ *num_kernel*. For each long live range i, we generate a random number, *length_i*, over [3,6] to represent the number of kernels spanned by i (longer live ranges should be split) and another random number over [1, *num_kernel-length_i*+1] to represent the kernel from which i starts to be live.

In Step 4, we keep $\mathcal{G}$ if $\mathcal{G}(V_l)$ is a forest and go back to Step 1 otherwise. In Step

5, we generate the stream sizes for all live ranges according to their characteristics in stream applications. In our experiments, node weights are chosen to have two different distributions, *Distribution* $\mathcal{U}$ and *Distribution* $\mathcal{L}$. Distribution $\mathcal{U}$, a discrete uniform distribution, in this case, node weights are randomly taken from the range [1,6]. For each program, we modify Distribution $\mathcal{U}$ to obtain Distribution $\mathcal{L}$ by simply replacing each stream size w by 2^w. In this second case, the fact that some streams may be geometrically larger than others in a program is explicitly taken into account. Distribution $\mathcal{L}$ is actually a uniform distribution in the logarithmic scale.

To test the scalability of FOREST_CGC, we have generated four groups of stream IGs. $G^1_{\mathcal{U}}$ and $G^1_{\mathcal{L}}$ consist of IGs with between 3 to 50 kernels with their node weights generated using Distributions $\mathcal{U}$ and $\mathcal{L}$, respectively. $G^2_{\mathcal{U}}$ and $G^2_{\mathcal{L}}$ consist of larger IGs with between 50 to 100 kernels. Each group consists of exactly 30 different IGs (with their node weights being ignored). For each IG in each group, there are 10 instances of that IG instantiated with different node weights (in Step 5). So each group consists of 300 different weighted IGs, giving rise to a total of 1200 stream IGs. Due to space limitation, we include only the results for Group $G^1_{\mathcal{L}}$ and Group $G^2_{\mathcal{U}}$ in Table II. The node counts of the graphs in these two groups are shown in Column 2 and Column 5, respectively, and their edge counts in Column 3 and Column 6, respectively. The tree counts in the forests $\mathcal{G}(V_l)$ in these graphs are shown in Column 4 and Column 7, respectively.

To check the optimality of FOREST_CGC, we have developed a formulation of the SRF allocation problem by integer linear programming (ILP). We ran the commercial ILP solver, CPLEX 10.1, to find an optimal coloring for each IG. If CPLEX does not terminate in five hours for an IG $\mathcal{G}$, its optimal coloring is estimated by (2). So all optimality results about FOREST_CGC reported here are conservative.



	$G^1_{\mathcal{L}}$	$G^1_{\mathcal{U}}$	$G^2_{\mathcal{L}}$	$G^2_{\mathcal{U}}$
FOREST_CGC	*299*	*298*	*296*	*295*
First_Fit_1	*262*	*189*	*281*	*191*
First_Fit_2	*55*	*115*	*38*	*104*

Fig. 18: Optimality of FOREST_CGC and First-Fit for 1200 IGs.

Figure 18 shows that FOREST_CGC obtains optimal solutions in 99% of the 1200 IGs in all four groups. In contrast, the solutions from First-Fit are mostly sub-optimal, with only 76.9% (First-Fit-1) and 26% (First-Fit-2) of the 1200 IGs being optimal. These results for Groups $G^1_{\mathcal{L}}$ and $G^2_{\mathcal{U}}$ are shown in Figures 19 and 20, respectively. In each figure, the x axis represents the 30 different IGs corresponds

to Column 1 in Table II and the y axis depicts the number of optimal solutions found by FOREST_CGC and First-Fit among the ten different IGs associated with each IG.

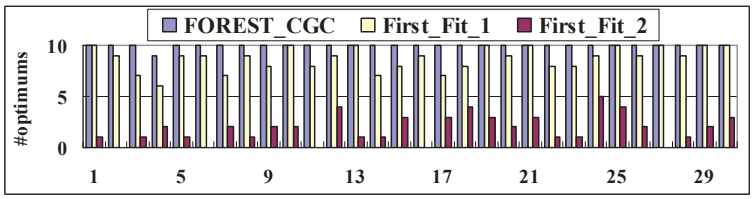


Fig. 19: Number of optimal solutions found by FOREST_CGC and First-Fit in 300 weighted IGs from $G^1_{\mathcal{L}}$.

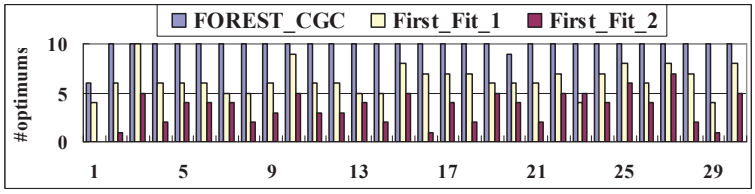


Fig. 20: Number of optimal solutions found by FOREST_CGC and First-Fit in 300 weighted IGs from $G^2_{\mathcal{U}}$.

The near-optimality of FOREST_CGC is achieved efficiently as validated in our experiments on a 3.2GHz Pentium 4 with 1GB memory. The longest time taken is 0.2 seconds for an IG with 409 nodes and 1836 edges, in which case $G(V_l)$ has 8 trees. For most of the other IGs, the times elapsed are less than 0.05 seconds each.

Let us look at the differences between FOREST_CGC and First-Fit in terms of their allocation results. For a given weighted IG $\mathcal{G}$, the quality of our solution α found by FOREST_CGC is measured as a *gap* with respect to that found by First-Fit defined as follows:

$$\mathsf{gap}(\mathcal{G}) \;=\; \frac{\chi_{\text{First–Fit}}(\mathcal{G};w) - \chi_{\alpha}(\mathcal{G};w)}{\chi_{\text{First–Fit}}(\mathcal{G};w)} \tag{7}$$

where $\chi_{\text{First–Fit}}(\mathcal{G};w)$ is the optimal solution (i.e., the smallest width required for coloring $\mathcal{G}$) found by First-Fit and $\chi_{\alpha}(\mathcal{G};w)$ is the optimal solution from FOREST_CGC.

Group	$< 0\%$	0%	$(0\%, 10\%)$	$[10\%, 20\%)$	$[20\%, 30\%)$
$G^1_{\mathcal{L}}$	1	262	29	7	1
$G^1_{\mathcal{U}}$	1	189	99	11	0
$G^2_{\mathcal{L}}$	3	278	16	3	0
$G^2_{\mathcal{U}}$	2	190	106	2	0

Table III: Gaps between FOREST_CGC and First-Fit-1 for 1200 weighted IGs.

Group	$G^1_{\mathcal{L}}$	$G^1_{\mathcal{U}}$	$G^2_{\mathcal{L}}$	$G^2_{\mathcal{U}}$
Mean gap	0.799%	1.883%	0.379%	1.352%

Table IV: Mean gap between FOREST_CGC and First-Fit-1.

Group	$< 0\%$	0%	$(0\%, 10\%)$	$[10\%, 20\%)$	$[20\%, 30\%)$
$G^1_{\mathcal{L}}$	1	55	175	61	8
$G^1_{\mathcal{U}}$	2	114	165	19	0
$G^2_{\mathcal{L}}$	2	38	217	40	3
$G^2_{\mathcal{U}}$	1	105	179	14	1

Table V: Gaps between FOREST_CGC and First-Fit-2 for 1200 weighted IGs.

Group	$G^1_{\mathcal{L}}$	$G^1_{\mathcal{U}}$	$G^2_{\mathcal{L}}$	$G^2_{\mathcal{U}}$
Mean gap	5.833%	3.079%	5.044%	3.533%

Table VI: Mean gap between FOREST_CGC and First-Fit-2.

Tables III and V show the gaps between FOREST_CGC and First-Fit for the four groups, $G^1_{\mathcal{L}}$, $G^1_{\mathcal{U}}$, $G^2_{\mathcal{L}}$ and $G^2_{\mathcal{U}}$, with 300 weighted graphs in each group. For Table V, for 69 out of 300 weighted graphs in Group $G^1_{\mathcal{L}}$, FOREST_CGC achieves a gap of over 10%. The largest for this set of 300 graphs is 29% for a graph consisting of 114 nodes and 455 edges. Finally, we observe that FOREST_CGC may perform slightly worse than First-Fit in six different weighted graphs. The gaps for five of these graphs are between -5.69% — -2.86% and the gap for the remaining one is -13.89%. Tables IV and VI demonstrate the mean gaps between FOREST_CGC and First-Fit for the 1200 weighted IGs from four groups. In general, the gaps depend on the distribution of the node weights, i.e., the sizes of streams in a program. The major advantage of FOREST_CGC is that if an IG is a comparability graph, then the optimal allocation is guaranteed regardless of what its node weights are, and in addition, even when an IG is not a comparability graph, FOREST_CGC can still achieve near-optimal colorings as proved in Theorems 4.11 and 4.12 and confirmed by the experimental data in Figure 18. However, the performance of First-Fit is sensitive to the structure of an IG and the values of its node weights. For example, the gap as shown in Figure 7 is 30%.

Let us also examine a counter example for which First-Fit-2 outperforms FOREST_CGC. This IG consists of 63 nodes. In addition, $\mathcal{G}(V_l)$ has one tree formed by nodes numbered 62 and 63. All the other nodes represent short live ranges. This IG is not a comparability graph. FOREST_CGC generates four acyclic orientations to the IG, as shown in Figures 21(a) – (d) (with some irrelevant nodes and edges omitted) – (a) and (d) are equivalent and (b) and (c) are equivalent. The heaviest directed path in each acyclic orientation is highlighted by bold arrows. The heaviest directed path in Figure 21(a) or (d) is $62, 51, 15, 16, 48, 49, 50, 63$, forming a clique (without node 62) with a total weight of 328. The heaviest directed path in Figures 21(b) or (c) is $39, 40, 41, 7, 8, 9, 42, 43, 44, 63$, forming a clique (without node 63) with a total weight of 336. So the optimal result achieved by FOREST_CGC is 328. In fact, the optimal orientation is shown in Figure 21(e). The heaviest directed

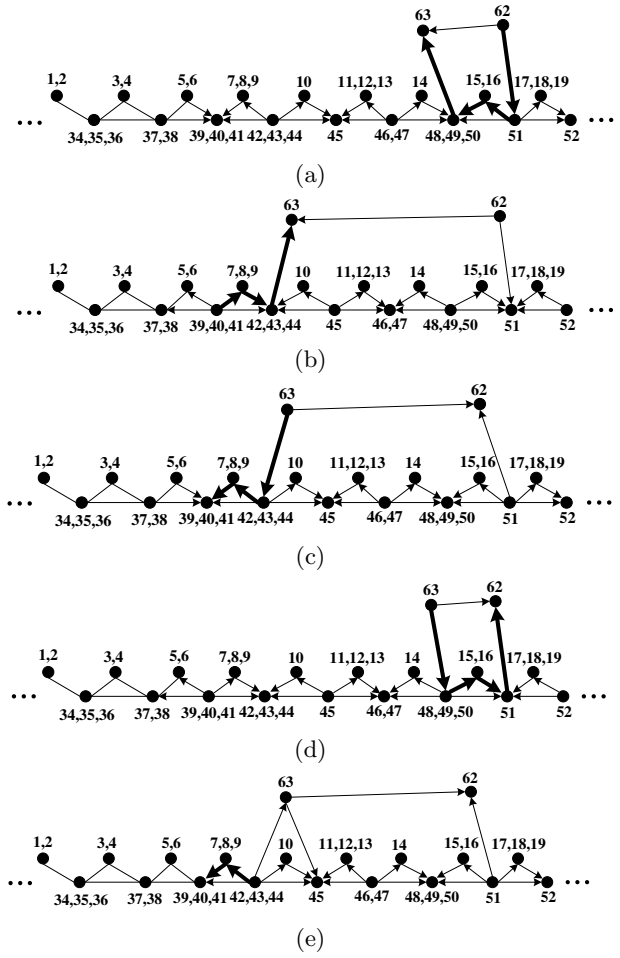


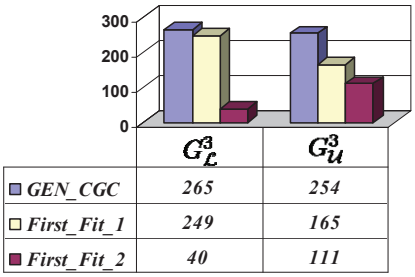
Fig. 21: A counter example.

path is $42, 43, 44, 7, 8, 9, 41, 40, 39$ with a total weight of 288. This is the heaviest directed path in Figures 21(b) or (c) with node 63 removed, corresponding to the situation identified in Theorem 4.12. For this IG, First-Fit accidently achieves the best coloring.

5.2 $\mathcal{G}(V_l)$ Is Not a Forest

In this case, GEN_CGC is invoked. In order to generate the IGs such that $\mathcal{G}(V_l)$ is not a forest, some minor changes are made to the algorithm used in Section 5.1 for generating random IGs. In step 4, when $\mathcal{G}(V_l)$ is not a forest, we keep rather than discard the generated IG. In addition, we extend the range of the random variable p from 1% to 30% to admit more long live ranges. We generate two groups of IGs in total. $G^3_{\mathcal{U}}$ and $G^3_{\mathcal{L}}$ consist of IGs with between 3 to 50 kernels. Their node weights are generated using Distributions $\mathcal{U}$ and $\mathcal{L}$, respectively. Each group consists of exactly 30 different IGs (with their node weights being ignored). For

each IG in each group, there are 10 instances of that IG instantiated with different node weights. So each group consists of 300 different weighted IGs, giving rise to a total of 600 IGs. Our experimental reports for these two groups are reported in Table II. The node counts of the graphs in these two groups are shown in Column 8 and Column 11 and the edge counts are shown in Column 9 and Column 12. The number of edges which are not in the spanning forest of $\mathcal{G}(V_l)$ are shown in Column 10 and Column 13.



	$G^3_{\mathcal{L}}$	$G^3_{\mathcal{U}}$
GEN_CGC	265	254
First_Fit_1	249	165
First_Fit_2	40	111

Fig. 22: Optimality of GEN_CGC and First-Fit for 600 IGs.

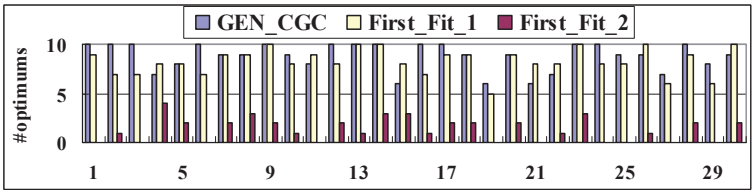


Fig. 23: Number of optimal solutions from GEN_CGC and First-Fit in 300 weighted IGs from $G^3_{\mathcal{L}}$.

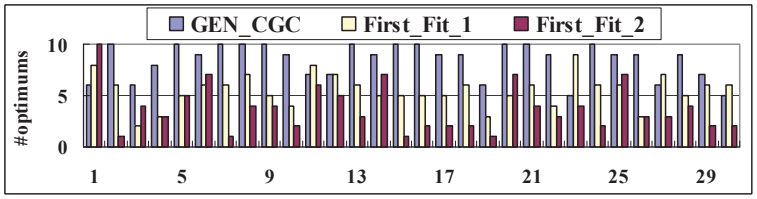


Fig. 24: Number of optimal solutions from GEN_CGC and First-Fit in 300 weighted IGs from $G^3_{\mathcal{U}}$.

Figure 22 shows that GEN_CGC obtains optimal solutions in 86.5% of the 600 IGs in two groups. In contrast, the solutions from First-Fit are mostly sub-optimal, with 69% (First-Fit-1) and 25.2% (First-Fit-2) of the 600 IGs being optimal. The detailed results are shown in Figures 23 and 24, respectively.

Tables VII and IX show the gaps between GEN_CGC and First-Fit for the two groups, $G^3_{\mathcal{L}}$ and $G^3_{\mathcal{U}}$, with 300 weighted graphs in each group. For Table IX, for 68 out of 300 IGs in Group $G^3_{\mathcal{L}}$, GEN_CGC achieves a gap of over 10%. Tables VIII

Group	$< 0\%$	0%	$(0\%, 10\%)$	$[10\%, 20\%)$
$G^3_{\mathcal{L}}$	29	228	36	7
$G^3_{\mathcal{U}}$	32	152	112	4

Table VII: Gaps between GEN_CGC and First-Fit-1 for 600 weighted IGs.

Group	$G^3_{\mathcal{L}}$	$G^3_{\mathcal{U}}$
Mean gap	0.185%	1.130%

Table VIII: Mean gap between GEN_CGC and First-Fit-1.

Group	$< 0\%$	0%	$(0\%, 10\%)$	$[10\%, 20\%)$	$[20\%, 30\%)$
$G^3_{\mathcal{L}}$	15	40	177	56	12
$G^3_{\mathcal{U}}$	21	108	152	19	0

Table IX: Gaps between GEN_CGC and First-Fit-2 for 600 weighted IGs.

and X demonstrate the mean gaps between GEN_CGC and First-Fit for the 600 weighted IGs from two groups.

6. RELATED WORK

Stream scheduling introduced in [Das et al. 2006] was earlier implemented in the StreamC compiler to compile stream programs for Imagine [Das et al. 2006; Owens et al. 2002]. Stream scheduling associates (if possible) all stream accesses to the same stream with the same buffer in the SRF. All such SRF buffers are placed in the SRF by applying some greedy First-Fit-like bin-packing heuristics. The key idea is trying to position each buffer at the smallest possible SRF address, always complete the current buffer before starting another, and finally, position the largest buffers first so that smaller buffers can fill in the cracks. In [Yang et al. 2008], we focus on identifying and representing loop-dependent reuse between streams.

This paper extends our previous work [Yang et al. 2009] in four ways. First, a general algorithm that works for any arbitrary stream IG is presented while our earlier algorithm is limited to stream IGs decomposable into comparability graphs and a forest. Second, more benchmarks and more experimental evaluation are included. Third, a counter example is presented showing First-Fit may occasionally achieve better colorings than our algorithm. Finally, all results are now rigorously proved.

Fabri [1979] recognized the connection between interval coloring and compile-time memory allocation. However, interval coloring is NP-complete even when restricted to interval graphs (a class of so-called perfect graphs) with vertex weights in $\{1, 2\}$ [Garey and Johnson 1979]. Since then, the research of applying interval coloring to compile-time memory allocation focuses on straight-line programs, i.e, programs without loops or conditional statements, in which case, their IGs are interval graphs. A number of approximation algorithms have been proposed [Kierstead 1988; 1991; Gergov 1996; 1999; Buchsbaum et al. 2003]. In particular, Kierstead [1988] presented the first constant-factor approximation algorithm, where the fac-

Group	$G^3_{\mathcal{L}}$	$G^3_{\mathcal{U}}$
Mean gap	5.690%	2.840%

Table X: Mean gap between GEN_CGC and First-Fit-2.

tor is 80. Later he reduced the factor to 6 [Kierstead 1991]. Subsequently, the factor was further reduced to 5 [Gergov 1996] and then to 3 [Gergov 1999]. Recently, Buchsbaum et al. [2003] has made further progress in reducing the factor to be $2 + \varepsilon$. However, despite these many years of continuous progress, the upper bound from mathematical analysis remains too conservative to be practically useful in computer science applications. Furthermore, straight-line programs (interval graphs) are too limited to be directly applied to real-world computer programs. In addition to compile-time memory allocation, Lefebvre and Feautrier [1998] use interval coloring to minimize the number of data structures to rename in storage management for parallel programs. Recently, Li et al. [2007] apply interval coloring to assign arrays in embedded programs to SPM.

Due to the one-to-one correspondence between interval colorings and acyclic orientations [Golumbic 2004], enumerating acyclic orientations for a given graph is another way to solve the interval coloring problem. Squire [1998] presented the first algorithm to generate all the acyclic orientations of an arbitrary graph. Barbosa and Szwarcfiter [1999] proposed another algorithm with lower complexity. [Stanley 1973] discovered that the number of acyclic orientations in a graph can be derived from its chromatic polynomial. Unfortunately, performing the acyclic orientation enumeration for an arbitrary graph is exponential [Linial 1986].

In [Wu et al. 2006], some improvements of [Das et al. 2006] to LRFs (Local Register Files) allocation in stream processor are presented. The LRFs are register files near the ALU clusters in a stream processor. Unlike SRF, LRFs play a similar role as the register files in general-purpose processors.

7. CONCLUSION

This paper presents a new approach to optimizing utilization of the SRF for stream processors. The key insight is that the interference graphs (IGs) in media and scientific applications amenable to stream processing are comparability graphs or decomposable into well-structured comparability subgraphs. This has motivated the development of a new algorithm that is capable of finding optimal or near-optimal colorings efficiently, thereby outperforming frequently used First-Fit heuristics.

Our IG-based algorithm allows other pre-pass optimizations such as live range splitting and stream prefetching to be well integrated into the same compiler framework. Finally, although developed for a non-bypassing SRF, our algorithm is applicable to any software-managed memory allocation for stream applications under a similar stream programming model.

ACKNOWLEDGMENTS

We wish to thank Yu Deng and Ying Zhang for some helpful discussions on this work. This research is supported in part by the National Natural Science Foundation of China (61003081), the Funds for Creative Research Groups of China

(60921062) and Australian Research Grants (DP0881330 and DP110104628).

REFERENCES

Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. 2002. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*. ACM, New York, NY, USA, 73–78.

Barbosa, V. C. and Szwarcfiter, J. L. 1999. Generating all the acyclic orientations of an undirected graph. *Inf. Process. Lett. 72,* 1-2, 71–74.

Buchsbaum, A. L., Karloff, H., Kenyon, C., Reingold, N., and Thorup, M. 2003. Opt versus load in dynamic storage allocation. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 556–564.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. 2004. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph. 23,* 3, 777–786.

Cuvillo, J., Zhu, W., Ziang, H., and Gao, G. 2005. Fast: A functionally accurate simulation toolset for the cyclops64 cellular architecture. In *In MoBS2005: Workshop on Modeling, Benchmarking, and Simulation*. ACM Press, 11–20.

Dally, W. J., Labonte, F., Das, A., Hanrahan, P., and et al, J.-H. A. 2003. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 35.

Das, A., Dally, W. J., and Mattson, P. 2006. Compiling for stream processing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, New York, NY, USA, 33–42.

Fabri, J. 1979. Automatic storage optimization. *SIGPLAN Not. 14,* 8, 83–91.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Gergov, J. 1996. Approximation algorithms for dynamic storage allocations. In *ESA '96: Proceedings of the Fourth Annual European Symposium on Algorithms*. Springer-Verlag, London, UK, 52–61.

Gergov, J. 1999. Algorithms for compile-time memory optimization. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 907–908.

Golumbic, M. C. 2004. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands.

Gummaraju, J., Coburn, J., Turner, Y., and Rosenblum, M. 2008. Streamware: programming general-purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, 297–307.

Gummaraju, J. and Rosenblum, M. 2005. Stream programming on general-purpose processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 343–354.

Kahn, A. B. 1962. Topological sorting of large networks. *Communications of the ACM 5,* 11, 558 – 562.

Kierstead, H. A. 1988. The linearity of first-fit coloring of interval graphs. *SIAM J. Discret. Math. 1,* 4, 526–530.

Kierstead, H. A. 1991. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Math. 87,* 2-3, 231–237.

Koch, K. 2006. the new roadrunner supercomputer: What, when, and how. In *Proceedings of International Conference on High Performance Computing*.

Kudlur, M. and Mahlke, S. 2008. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 114–124.

Labonte, F., Mattson, P., Thies, W., Buck, I., Kozyrakis, C., and Horowitz, M. 2004. The stream virtual machine. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. 267–277.

Lefebvre, V. and Feautrier, P. 1998. Automatic storage management for parallel programs. *Parallel Comput. 24,* 3-4, 649–671.

Leverich, J., Arakida, H., Solomatnikov, A., Firoozshahian, A., Horowitz, M., and Kozyrakis, C. 2007. Comparing memory systems for chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, New York, NY, USA, 358–368.

Li, L., Nguyen, Q. H., and Xue, J. 2007. Scratchpad allocation for data aggregates in superperfect graphs. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. ACM, 207–216.

Linial, N. 1986. Hard enumeration problems in geometry and combinatorics. *SIAM J. Algebraic Discrete Methods 7,* 2, 331–335.

Makino, J., Hiraki, K., and Inaba, M. 2007. Grape-dr: 2-pflops massively-parallel computer with 512-core, 512-gflops processor chips for scientific computing. In *In SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 1C11.

Murthy, P. K. and Bhattacharyya, S. S. 2004. Buffer merging a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Trans. Des. Autom. Electron. Syst. 9*, 212–237.

Owens, J. D., Kapasi, U. J., Mattson, P., Towles, B., Serebrin, B., Rixner, S., and Dally, W. J. 2002. Media processing applications on the imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design*. 295–302.

Rixner, S., Dally, W. J., Kapasi, U. J., Khailany, B., Lpez-Lagunas, A., Mattson, P. R., and Owens, J. D. 1998. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*. 3–13.

Squire, M. B. 1998. Generating the acyclic orientations of a graph. *J. Algorithms 26,* 2, 275–290.

Stanley, R. P. 1973. Acyclic orientations of graphs. *Discrete Math 5*, 171–178.

Taylor, M., Kim, J., and Miller, J. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro 22,* 2, 25–35.

Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., Ho, H., Brown, M., and Amarasinghe, S. 2001. StreamIt: A compiler for streaming applications. MIT-LCS Technical Memo TM-622.

Viterbi, A. J. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Information Theory IT-13*, 260–269.

Wang, L., Yang, X., Xue, J., Deng, Y., Yan, X., Tang, T., and Nguyen, Q. H. 2008. Optimizing scientific application loops on stream processors. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. ACM, 161–170.

West, D. B. 1996. *Introduction To Graph Theory*. Prentice Hall.

Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. 2006. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*. ACM, New York, NY, USA, 9–20.

Wu, N., Wen, M., Ren, J., He, Y., and Zhang, C. 2006. Register allocation on stream processor with local register file. In *ACSAC '06: Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference*. 545–551.

Yang, X., Wang, L., Xue, J., Deng, Y., and Zhang, Y. 2009. Comparability graph coloring for optimizing utilization of stream register files in stream processors. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, New York, NY, USA, 111–120.

Yang, X., Yan, X., Xing, Z., Deng, Y., Jiang, J., and Zhang, Y. 2007. A 64-bit stream processor architecture for scientific applications. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 210–219.

Yang, X., Zhang, Y., Xue, J., Rogers, I., Li, G., and Wang, G. 2008. Exploiting loop-dependent stream reuse for stream processors. In *PACT '08: The Seventeenth International Conference on Parallel Architectures and Compilation Techniques*.