In EURASIP Journal on Applied Signal Processing, 2002(9):893-907, September 2002.

High-Level Synthesis of DSP Applications using Adaptive Negative Cycle Detection*

Nitin Chandrachoodan, Shuvra S. Bhattacharyya†† and K. J. Ray Liu
Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD 20742
(nitin,ssb,kjrliu@eng.umd.edu)

Abstract

Detection of negative cycles in a weighted directed graph is a problem that plays an important role in the high-level synthesis (HLS) of DSP applications. In this paper, this problem is examined in the context of the "dynamic graph" structures that arise in the process of HLS.

The concept of *adaptive* negative cycle detection is introduced, in which a graph changes over time and negative cycle detection needs to be done periodically, but not necessarily after every individual change. Such scenarios arise, for example, during iterative design space exploration for hardware and software synthesis. We present an algorithm for this problem, based on a novel extension of the well known Bellman-Ford algorithm that allows us to adapt our existing cycle information to the modified graph, and show by experimental results that our algorithm significantly outperforms previous approaches for dynamic graphs, which do not take advantage of multiple simultaneous changes to the graph and require excessive computation.

The adaptive technique introduced here is used to solve two important problems in the HLS process: performance analysis and system synthesis. The adaptive technique leads to a very fast implementation of Lawler's algorithm for the computation of the maximum-cycle mean (MCM) of a graph, especially for a certain form of *sparse graph*. Such sparseness often occurs in practical circuits and systems, as demonstrated for example by the ISCAS 89/93 benchmarks. The application of the adaptive technique to design-space exploration (synthesis) is also demonstrated by developing automated search techniques for scheduling iterative data-flow graphs.

*This research was supported in part by the US National Science Foundation Grant #9734275 and NSF NYI Award MIP9457397.

††Also with the University of Maryland Institute for Advanced Computer Studies.

1 Introduction

High-Level synthesis of circuits for Digital Signal Processing (DSP) applications is an area of considerable interest due to the rapid increase in the number of devices requiring multimedia and DSP algorithms. High-level synthesis (HLS) plays an important role in the overall system synthesis process because it speeds up the process of converting an algorithm into an implementation in hardware, software, or a mixture of both. In HLS, the algorithm is represented in an abstract form (usually a dataflow graph), and this representation is transformed and mapped onto architectural elements from a library of resources. These resources could be pure hardware elements like adders or logic gates, or they could be general purpose processors with the appropriate software to execute the required operations. The architecture could also involve a combination of both the above, in which case the problem becomes one of hardware-software co-synthesis.

HLS involves several stages and requires computation of several parameters. In particular, performance estimation is one very important part of the HLS process, and the actual design space exploration (synthesis of architecture) is another. Performance estimation involves using timing information about the library elements to obtain an estimate of the throughput that can be obtained from the synthesized implementation. A particularly important estimate for iterative dataflow graphs is known as the Maximum Cycle-Mean (MCM) [25, 11]. This quantity provides a bound on the maximum throughput attainable by the system. A fast method for computing the MCM would therefore enable this metric to be computed for a large number of system configurations easily.

The other important problem in HLS is the problem of design space exploration. This requires the selection of an appropriate set of elements from the resource library and mapping of the dataflow graph functions onto these resources. This problem is known to be NP-complete, and there exist several heuristic approaches that attempt to provide sufficiently good solutions. An important feature of HLS of DSP applications is that the mapping to hardware needs to be done only once for a given design, which is then produced in large quantities for a consumer market. As a result, for these applications (such as modems, wireless phones, multimedia terminals etc.) it makes sense to consider the possibility of investing large amounts of computational power at "compile time", so that a more optimal result can be used at "run time".

The problems in HLS described above both have the common feature of requiring a fast solution to the problem of detecting negative cycles in a graph. This is because the execution times of the various resources combine with the graph structure to impose a set of constraints on the system, and checking the feasibility of this set of constraints is equivalent to checking for the presence of negative cycles in the corresponding constraint graph.

DSP applications, more than other embedded applications considered in HLS, have the property that they are "cyclic" in nature. As explained in the next section, this means that the problem of negative cycle detection in constraint analysis is more relevant to such systems. In order to make use of the increased computational power that is available, one possibility is to conduct more extensive searches of the design space than is performed by a single heuristic. One possible approach to this problem involves an iterative improvement system based on generating modified versions of an existing implementation and verifying their correctness. The incremental improvements can then be used to guide a search of the design space that can be tailored to fit in the maximum time allotted to the exploration problem. In this process, the most computationally intensive part is the process of verifying correctness of the modified systems, and therefore speeding up this process would have a direct impact on the size of the explored region of the design space.

In addition to these problems from HLS, several other problems in circuits and systems theory require the solving of constraint equations [7, 24, 11, 14, 19]. Examples include VLSI layout compaction, interactive (reactive) systems, graphic layout heuristics, and timing analysis and retiming of circuits for performance or area considerations. Though a general system of constraints would require a linear programming (LP) approach to solve it, several problems of interest actually consist of the special case of difference constraints (each constraint expresses the minimum or maximum value that the difference of two variables in the system can take). These problems can be attacked by faster techniques than the general LP, mostly involving the solution of a shortest path problem on a weighted directed graph. Detection of negative cycles in the graph is therefore a closely related problem, as it would indicate in-feasibility of the constraint system.

Because of the above reasons, detecting the presence of negative cycles in a weighted directed graph is a very important problem in systems theory. This problem is also important in the computation of network flows. Considerable effort has been spent on finding efficient algorithms for this purpose. Cherkassky and Goldberg [7] have performed a comprehensive survey of existing techniques. Their study shows some interesting features of the available algorithms, such as the fact that for a large class of random graphs, the worst case performance bound is far more pessimistic than the observed performance.

There are also situations in which it is useful or necessary to maintain a feasible solution to a set of difference constraints as a system evolves. Typical examples of this would be real-time or interactive systems, where constraints are added or removed one (or several) at a time, and after each such modification it is required to determine whether the resulting system has a feasible solution and if so, to find it. In these situations, it is often more efficient to adapt existing information to aid the solution of the constraint system. In the example from HLS that was mentioned previously, it is possible to cast the problem of design space exploration in a way that benefits from this approach.

Several researchers[22, 14, 2] have worked on the area of *incremental computation*. They have presented analyses of algorithms for the shortest path problem and negative cycle detection in *dynamic* graphs. Most of the approaches try to apply modifications of Dijkstra's algorithm to the problem. The obvious reason for this is that this is the fastest known algorithm for the problem when only positive weights are allowed on edges. However, use of Dijkstra's algorithm as the basis for incremental computation requires the changes to be handled one at a time. While this may often be efficient enough, there are many cases where the ability to handle multiple changes simultaneously would be more advantageous. For example, it is possible that in a sequence of changes, one reverses the effect of another: in this case, a normal incremental approach would perform the same computation twice, while a delayed adaptive computation would not waste any effort.

In this paper, we present an approach that generalizes the adaptive approach beyond single increments: we address multiple changes to the graph simultaneously. Our approach can be applied to cases where it is possible to collect several changes to the graph structure before updating the solution to the constraint set. As mentioned previously, this can result in increased efficiency in several important problems. We present simulation results comparing our method against the single-increment algorithm proposed in [24]. For larger numbers of changes, our algorithm performs considerably better than this incremental algorithm.

To illustrate the advantages of our adaptive approach, we present two applications from the area of HLS, requiring the solution of difference constraint problems, which therefore benefit from the application of our technique. For the problem of performance estimation, we show how the new technique can be used to derive a fast implementation of Lawler's algorithm for the problem of computing the maximum cycle-mean (MCM) of a weighted directed graph. We present experimental results comparing this against Howard's algorithm [8, 11], which appears to be the fastest algorithm available in practice. We find that for graph sizes and node-degrees similar to those of real circuits, our algorithm often outperforms even Howard's algorithm.

For the problem of design space exploration, we present a search technique for finding schedules for iterative dataflow graphs that uses the adaptive negative cycle detection algorithm as a subroutine. We illustrate the use of this local search technique by applying it to the problem of resource-constrained scheduling for minimum power in the presence of functional units that can operate at multiple voltages. The method we develop is quite general and can therefore easily be extended to the case of optimization where we are interested in optimizing other criteria rather than the power consumption.

Section 2 surveys previous work on shortest path algorithms and incremental algorithms. In Section 3, we describe the adaptive algorithm that works on multiple changes to a graph efficiently. Section 4 compares our algorithm against the existing approach, as well as against another possible candidate for adaptive operation. Section 5 then gives details of the applications mentioned above, and presents some experimental results. Finally, we present our conclusions and examine areas that would be suitable for further investigation.

A preliminary version of the results presented in this paper were published in [5].

2 Background and Problem Formulation

Cherkassky and Goldberg [7] have conducted an extensive survey of algorithms for detecting negative cycles in graphs. They have also performed a similar study on the problem of shortest path computations. They present several problem families that can be used to test the effectiveness of a cycle-detection algorithm. One surprising fact is that the best known theoretical bound ($O(|V||E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph) for solving the shortest path problem (with arbitrary weights) is also the best known time bound for the negative-cycle problem. But examining the experimental results from their work reveals the interesting fact that in almost all of the studied samples, the performance is considerably less costly than would be suggested by the product ($|V| \times |E|$). It appears that the worst case is rarely encountered in random examples, and an average case analysis of the algorithms might be more useful.

Recently, there has been increased interest in the subject of *dynamic* or *incremental* algorithms for solving problems[22, 2, 14]. This uses the fact that in several problems where a graph algorithm such as shortest paths or transitive closure needs to be solved, it is often the case that we need to repeatedly solve the problem on variants of the original graph. The algorithms therefore store information about the problem that was obtained during a previous iteration and use this as an efficient starting point for the new problem instance corresponding to the slightly altered graph. The concept of *bounded incremental computation* introduced in [22] provides a framework within which the improvement afforded by this approach can be quantified and analyzed.

In this paper, the problem we are most interested in is that of maintaining a solution to a set of difference constraints. This is equivalent to maintaining a shortest path tree in a dynamic graph [24]. Frigioni *et al.* [14] present an algorithm for maintaining shortest paths in arbitrary graphs that performs better than starting from scratch, while Ramalingam and Reps [23] present a generalization of the shortest path problem, and show how it can be used to handle the case where there are few negative weight edges. In both of these cases, they have considered one change at a time (not multiple changes), and the emphasis has been on the theoretical time bound, rather than experimental analysis. In [13], the authors present an experimental study, but only for the case of positive weight edges, which restricts the study to computation of shortest paths and does not consider negative weight cycles.

The most significant work along the lines we propose is described in [24]. In this, the authors use the observation that in order to detect negative cycles, it is not *necessary* to maintain a tree of the shortest paths to each vertex. They suggest an improved algorithm based on Dijkstra's algorithm, which is able to recompute a feasible solution (or detect a negative cycle) in time $O(E + V \log V)$, or in terms of *output complexity* (defined and motivated in [24]) $O(||\Delta|| + |\Delta| \log |\Delta|)$, where$|\Delta|$ is the number of variables whose values are changed and $||\Delta||$ is the number of constraints involving the variables whose values have changed.

The above problem can be generalized to allow multiple changes to the graph between calls to the negative cycle detection algorithm. In this case, the above algorithms would require the changes to be handled one at a time, and therefore would take time proportional to the total number of changes. On the other hand, it would be preferable if we could obtain a solution whose complexity depends instead on the number of *updates* requested, rather than the total number of changes applied to the graph. Multiple changes between updates to the negative cycle computation arise naturally in many interactive environments, (*e.g.,* if we prefer to accumulate changes between refreshes of the state, using the idea of lazy evaluation) or in design space-exploration, as can be seen, for example, in section 5.2. By accumulating changes and processing them in large batches, we remove a large overhead from the computation, which may result in considerably faster algorithms.

Note that the work in [24] also considers the addition/deletion of constraints only one at a time. It needs to be emphasized that this limitation is basic to the design of the algorithm: Dijkstra's algorithm can be applied only when the changes are considered one at a time. This is acceptable in many contexts since Dijkstra's algorithm is the fastest algorithm for the case where edge weights are positive. If we try using another shortest-paths algorithm we would incur a performance penalty. However, as we show, this loss in performance in the case of unit changes may be offset by improved performance when we consider multiple

changes.

The approach we present for the solution is to extend the classical Bellman-Ford algorithm for shortest paths in such a way that the solution obtained in one problem instance can be used to reduce the complexity of the solution in modified versions of the graph. In the incremental case (single changes to the graph) this problem is related to the problem of analyzing the "sensitivity" of the algorithm [1]. The sensitivity analysis tries to study the performance of an algorithm when its inputs are slightly perturbed. Note that there do not appear to be any average case sensitivity analyses of the Bellman-Ford algorithm, and the approach presented in [1] has a quadratic running time in the size of the graph. This analysis is performed for a general graph without regard to any special properties it may have. But as explained in sec. 5.1.1, graphs corresponding to circuits and systems in HLS for DSP are typically very sparse – most benchmark graphs tend to have a ratio of about 2 edges per vertex, and the number of delay elements is also small relative to the total number of vertices. Our experiments have shown that in these cases, the adaptive approach is able to do much better than a quadratic approach. We also provide application examples to show other potential uses of the approach.

In the following sections, we show that our approach performs almost as well as the approach in [24] (experimentally) for changes made one at a time, and significantly outperforms their approach under the general case of multiple changes (this is true even for relatively small batches of changes, as will be seen from the results). Also, when the number of changes between updates is very large, our algorithm reduces to the normal Bellman-Ford algorithm (starting from scratch), so we do not lose in performance. This is important since when a large number of changes are made, the problem can be viewed as one of solving the shortest-path problem for a new graph instance, and we should not perform worse than the standard available technique for that.

Our interest in adaptive negative cycle detection stems primarily from its application in the problems of HLS that we outlined in the introduction. To demonstrate its usefulness in these areas, we have used this technique to obtain improved implementations of the performance estimation problem (computation of the MCM) and to implement an iterative improvement technique for design space exploration. Dasdan *et al.* [11] present an extensive study of existing algorithms for computing the MCM. They conclude that the most efficient algorithm in practice is Howard's algorithm [8]. We show that the well known Lawler's algorithm [18], when implemented using an efficient negative-cycle detection technique and with the added benefit of our adaptive negative cycle detection approach, actually outperforms this algorithm for several test cases, including several of the ISCAS benchmarks, which represent reasonable sized circuits.

As mentioned previously, the relevance of negative cycle detection to design space exploration is because of the cyclic nature of the graphs for DSP applications. That is, there is often a dependence between the computation in one iteration and the values computed in previous iterations. Such graphs are referred to as "Iterative dataflow graphs" [12]. Traditional scheduling techniques tend to consider only the latency of the system, converting it to an acyclic graph if necessary. This can result in loss of the ability to exploit inter-iteration parallelism effectively. Methods such as "Optimum unfolding" [21] and "Range-chart guided scheduling" [12] are techniques that try to avoid this loss in potential parallelism by working directly on the cyclic graph. However, they suffer from some disadvantages of their own. Optimum unfolding can potentially lead to a large increase in the size of the resulting graph to be scheduled. Range chart guided scheduling is a deterministic heuristic that could miss potential solutions. In addition, the process of scanning through all possible time intervals for scheduling an operation can work only when the run-times of operations are small integers. This is more suited to a software implementation than a general hardware design. These techniques also work only after a function to resource binding is known, as they require timing information for the functions in order to schedule them. For the general architecture synthesis problem, this binding itself needs to be found through a search procedure, so it is reasonable to consider alternate search schemes that combine the search for architecture with the search for a schedule.

If the cyclic dataflow graph is used to construct a constraint graph, then feasibility of the resulting system is determined by the absence of negative cycles in the graph. This can be used to obtain exact schedules capable of attaining the performance bound for a given function to resource binding. For the problem of design space exploration, we treat the problem of scheduling an iterative dataflow graph (IDFG)

as a problem of searching for an efficient ordering of function vertices on processors, which can be treated as addition of several timing constraints to an existing set of constraints. We implement a simple search technique that uses this approach to solve a number of scheduling problems, including scheduling for low-power on multiple-voltage resources, and scheduling on homogeneous processors, within a single framework. Since the feasibility analysis forms the core of the search, speeding this up should result in a proportionate increase in the number of designs evaluated (until such a point that this is no longer the bottleneck in the overall computation). The adaptive negative cycle detection technique ensures that we can do such searches efficiently, by restricting the computations required.

3 The Adaptive Bellman-Ford Algorithm

In this section, we present the basis of the adaptive approach that enables efficient detection of negative cycles in dynamic graphs.

We first note that the problem of detecting negative cycles in a weighted directed graph (digraph) is equivalent to finding whether or not a set of difference inequality constraints has a feasible solution. To see this, observe that if we have a set of difference constraints of the form

$$x_i - x_j \leq b_{ij}$$

we can construct a digraph with vertices corresponding to the x_i, and an edge (e_{ij}) directed from the vertex corresponding to x_i to the vertex for x_j such that $weight(e_{ij}) = b_{ij}$. This procedure is performed for each constraint in the system and a weighted directed graph is obtained. Solving for shortest paths in this graph would yield a set of distances *dist* that satisfy the constraints on x_i. This graph is henceforth referred to as the *constraint graph*.

The usual technique used to solve for *dist* is to introduce an imaginary vertex s_0 to act as a source, and introduce edges of zero-weight from this vertex to each of the other vertices. The resulting graph is referred to as the *augmented graph* [24]. In this way, we can use a single-source shortest paths algorithm to find *dist* from s_0, and any negative cycles (infeasible solution) found in the augmented graph must also be present in the original graph, since the new vertex and edges cannot create cycles.

The basic Bellman-Ford algorithm does not provide a standard way of detecting negative cycles in the graph. However, it is obvious from the way the algorithm operates that if changes in the distance labels continue to occur for more than a certain number of iterations, there must be a negative cycle in the graph. This observation has been used to detect negative cycles, and with this straightforward implementation, we obtain an algorithm to detect negative cycles that takes $O(|V|^3)$ time, where $|V|$ is the number of vertices in the graph.

The study by Cherkassky and Goldberg [7] presents several variants of the negative cycle detection technique. The technique they found to be most efficient in practice is based on the "subtree disassembly" technique proposed by Tarjan. This algorithm works by constructing a shortest path tree as it proceeds from the source of the problem, and any negative cycle in the graph will first manifest itself as a violation of the tree order in the construction. The experimental evaluation presented in their study found this algorithm to be a robust variant for the negative cycle detection problem. As a result of their findings, we have chosen this algorithm as the basis for the adaptive algorithm. Our modified algorithm is henceforth referred to as the "Adaptive Bellman-Ford (ABF)" algorithm.

The adaptive version of the Bellman-Ford algorithm works on the basis of storing the distance labels that were computed from the source vertex from one iteration to the next. Since the negative cycle detection problem requires that the source vertex is always the same (the augmenting vertex), it is intuitive that as long as most edge weights do not change, the distance labels for most of the vertices will also remain the same. Therefore, by storing this information and using it as a starting point for the negative cycle detection routines, we can save a considerable amount of computation.

One possible objection to this system is that we would need to scan all the edges each time in order to detect vertices that have been affected. But in most applications involving multiple changes to a graph, it

is possible to pass information to the algorithm about which vertices have been affected. This information can be generated by the higher level application-specific process making the modifications. For example, if we consider multiprocessor scheduling, the high-level process would generate a new vertex ordering, and add edges to the graph to represent the new constraints. Since any changes to the graph can only occur at these edges, the application can pass on to the ABF algorithm precise information about what changes have been made to the graph, thus saving the trouble of scanning the graph for changes.

Note that in the event where the high-level application cannot pass on this information without adding significant bookkeeping overhead, the additional work required for a scan of the edges is proportional to the number of edges, and hence does not affect the overall complexity, which is at least as large as this. For example, in the case of the maximum cycle mean computation examined below, for most circuit graphs the number of edges with delays is about 1/10 as many as the total number of edges. With each change in the target iteration period, most of these edges will cause constraint violations. In such a situation, an edge scan provides a way of detecting violations that is very fast and easy to implement, while not increasing the overall complexity of the method.

3.1 Correctness of the method

The use of a shortest path routine to find a solution to a system of difference constraint equations is based on the following two theorems, which are not hard to prove (see [9]).

THEOREM 1 *A system of difference constraints is consistent if and only if its augmented constraint graph has no negative cycles, and the latter condition holds if and only if the original constraint graph has no negative cycles.*

THEOREM 2 *Let G be the augmented constraint graph of a consistent system of constraints $\langle V, C\rangle$. Then D is a feasible solution for $\langle V, C\rangle$, where*

$$D(u) = dist_G(s_0, u)$$

In Theorem 2, the *constraint graph* is defined as in sec. 3 above. The *augmented constraint graph* consists of this graph, together with an additional source vertex (s_0) that has 0-weight edges leading to all the other existing vertices, and *consistency* means that a set of x_i exist that satisfy all the constraints in the system.

In the adaptive version of the algorithm, we are effectively setting the weights of the augmenting edges to be equal to the labels that were computed in the previous iteration. In this way, the initial scan from the augmenting vertex sets the distance label at each vertex equal to the previously computed weight instead of setting it to 0. So we now need to show that using non-zero weights on the augmenting edges does not change the solution space in any way: *i.e.* all possible solutions for the 0-weight problem are also solutions for the non-zero weight problem, except possibly for translation by a constant.

The new algorithm with the adaptation enhancements can be seen to be correct if we relax the definition of the augmented graph so that the augmenting edges (from s_0) need not have 0 weight. We summarize the arguments for this in the following theorems:

THEOREM 3 *Consider a constraint graph augmented with a source vertex s_0, and edges from this vertex to every other vertex v, such that these augmenting edges have arbitrary weight $weight(s_0 \rightarrow v)$. The associated system of constraints is consistent if and only if the augmenting graph defined above has no negative cycles, which in turn holds if and only if the original constraint graph has no negative cycles.*

Proof: Clearly, since s_0 does not have any in-edges, no cycles can pass through it. So any cycles, negative or otherwise, which are detected in the augmented graph, must have come from the original constraint graph, which in turn would happen only if the constraint system was inconsistent (by Theorem 1). Also, any inconsistency in the original system would manifest itself as a negative cycle in the constraint graph, and the above augmentation cannot remove any such cycle. □

The following theorem establishes the validity of solutions computed by the ABF algorithm.

THEOREM 4 *If G' is the augmented graph with arbitrary weights as defined above, and $D(u) = dist_{G'}(s_0, u)$ (shortest paths from s_0), then*

1. *D is a solution to $\langle V, C\rangle$; and*
2. *Any solution to $\langle V, C\rangle$ can be converted into a solution to the constraint system represented by G' by adding a constant to $D(u)$ for each $u \in V$.*

Proof: The first part is obvious, by the definition of shortest paths.

Now we need to show that by augmenting the graph with arbitrary weight edges, we do not prevent certain solutions from being found. To see this, first note that any solution to a difference constraint system remains a solution when translated by a constant. That is, we can add or subtract a constant to all the $D(u)$ without changing the validity of the solution.

In our case, if we have a solution to the constraint system that does not satisfy the constraints posed by our augmented graph, it is clear that the constraint violation can only be on one of the augmenting edges (since the underlying constraint graph is the same as in the case where the augmenting edges had zero weight). Therefore, if we define

$$l_{max} = \max\{weight(e) | e \in S_a\},$$

where S_a is the set of augmenting edges and

$$D'(u) = D(u) - l_{max},$$

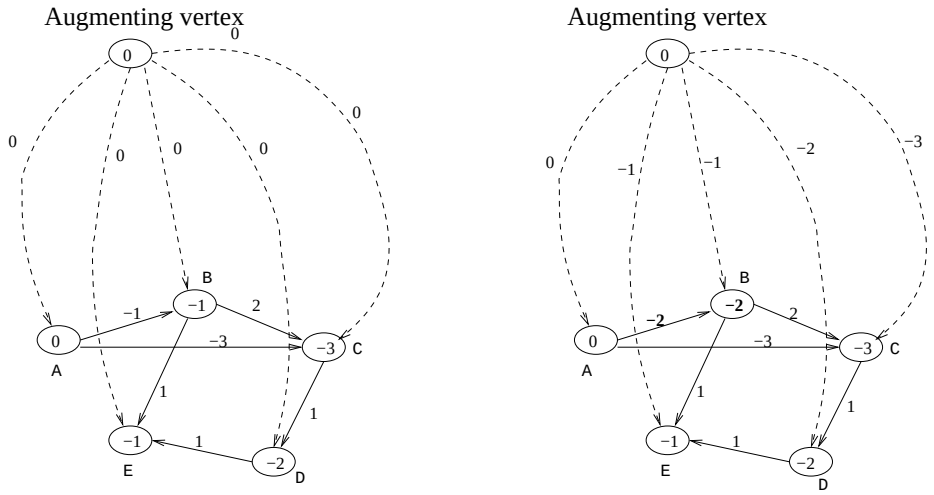
we ensure that D' satisfies all the constraints of the original graph, as well as all the constraints on the augmenting edges. □

Theorem 4 tells us that an augmented constraint graph with arbitrary weights on the augmenting edges can also be used to find a feasible solution to a constraint system. This means that once we have found a solution $dist : V \to \mathcal{R}$ (where $\mathcal{R}$ is the set of real numbers) to the constraint system, we can change the augmented graph so that the weight on each edge $e : u \to v$ is $dist(v)$. Now even if we change the underlying constraint graph in any way, we can use the same augmented graph to test the consistency of the new system.

Figure 1 helps to illustrate the concepts that are explained in the previous paragraphs. In part (B) of the figure, there is a change in the weight of one edge. But as we can see from the augmented graph, this will result in only the single update to the affected vertex itself, and all the other vertices will get their constraint satisfying values directly from the previous iteration.

Note that in general several vertices could be affected by the change in weight of a single edge. For example, in the figure, if edge AC had not existed, then changing the weight of AB would have resulted in a new distance label for vertices C and D as well. These would be cascading effects from the change in the distance label for vertex B. Therefore, when we speak of affected vertices, it is not just those vertices incident on an edge whose weight has changed, but could also consist of vertices not directly on an edge that has undergone a change in constraint weight. The actual number of vertices affected by a single edge-weight change cannot be determined just by examining the graph, we would actually need to run through the Bellman-Ford algorithm to find the complete set of vertices that are affected.

In the example from figure 1, the change in weight of edge AB means that after an initial scan to determine changes in distance labels, we find that vertex B is affected. However, on examining the outgoing edges from vertex B, we find that all other constraints are satisfied, so the Bellman-Ford algorithm can terminate here without proceeding to examine all other edges. Therefore, in this case, there is only 1 vertex whose label is affected out of the 5 vertices in the graph. Furthermore, the experiments show that even in large sparse graphs, the effect of any single change is usually localized to a small region of the graph, and this is the main reason that the adaptive approach is useful, as opposed to other techniques that are developed for more general graphs. Note that, as explained in the previous section, the initial overhead for detecting constraint violations still holds, but the complexity of this operation is significantly less than that of the Bellman-Ford algorithm.



(A) Augmenting graph with 0 wt. augmenting edges.

(B) Augmenting graph with non−zero weight augmenting edges.

Figure 1: Constraint graph.

4 Comparison against Other Incremental Algorithms

We compare the `ABF` algorithm against (a) the incremental algorithm developed in [24] for maintaining a solution to a set of difference constraints (referred to here as the `RSJM` algorithm), and (b) a modification of Howard's algorithm [8], since it appears to be the fastest known algorithm to compute the cycle mean, and hence can also be used to check for feasibility of a system. Our modification allows us to use some of the properties of adaptation to reduce the computation in this algorithm.

The main idea of the adaptive algorithm is that it is used as a routine inside a loop corresponding to a larger program. As a result, in several applications where this negative cycle detection forms a computation bottleneck, there will be a proportional speedup in the overall application which would be much larger than the speedup in a single run.

It is worth making a couple of observations at this point regarding the algorithms we compare against.

1. The `RSJM` algorithm [24] uses Dijkstra's algorithm as the core routine for quickly recomputing the shortest paths. Using the Bellman-Ford algorithm here (even with Tarjan's implementation) would result in a loss in performance since it cannot match the performance of Dijkstra's algorithm when edge weights are positive. Consequently, no benefit would be derived from the reduced-cost concept used in [24].

2. The code for Howard's algorithm was obtained from the Internet web-site of the authors of [8]. The modifications suggested by Dasdan *et al.* [10] have been taken into account. This method of constraints checking uses Howard's algorithm to see if the MCM of the system yields a feasible value, otherwise the system is deemed inconsistent.

Another important point is the type of graphs on which we have tested the algorithms. We have restricted our attention to *sparse* graphs, or *bounded degree* graphs. In particular, we have tried to keep the vertex-to-edge ratio similar to what we may find in practice, as in, for example, the ISCAS benchmarks. To understand why such graphs are relevant, note the following two points about the structural elements usually found in circuits and signal processing blocks: (a) they typically have a small, finite number of inputs and outputs (*e.g.* `AND` gates, adders, etc. are binary elements) and (b) the fanout that is allowed in these systems is usually limited for reasons of signal strength preservation (buffers are used if necessary). For these reasons,

the graphs representing practical circuits can be well approximated by bounded degree graphs. In more general DSP application graphs, constraints such as fanout may be ignored, but the modular nature of these systems (they are built up of simpler, small modules) implies that they normally have small vertex degrees.

We have implemented all the algorithms under the LEDA [20] framework for uniformity. The tests were run on random graphs, with several random variations performed on them thereafter. We kept the number of vertices constant and changed only the edges. This was done for the following reason: a change to a node (addition/deletion) may result in several edges being affected. In general, due to the random nature of the graph, we cannot know in advance the exact number of altered edges. Therefore, in order to keep track of the exact number of changes, we applied changes only to the edges. Note that when node changes are allowed, the argument for an adaptive algorithm capable of handling multiple changes naturally becomes stronger.

In the discussion that follows, we use the term "batch-size" to refer to the number of changes in a multiple change update. That is, when we make multiple changes to a graph between updates, the changes are treated as a single batch, and the actual number of changes that was made is referred to as the batch-size. This is a useful parameter to understand the performance of the algorithms.

The changes that were applied to the graph were of 3 types:

- *Edge insertion:* An edge is inserted into the graph, ensuring that multiple edges between vertices do not occur.
- *Edge deletions:* An edge is chosen at random and deleted from the graph. Note that, in general, this cannot cause any violations of constraints.
- *Edge weight change:* An edge is chosen at random and its weight is changed to another random number.

Figure 2 shows a comparison of the running time of the 3 algorithms on random graphs. The graphs in question were randomly generated, had 1,000 vertices and 2,000 edges each, and a sequence of 10,000 edge change operations (as defined above) were applied to them. The points in the plot correspond to an average over 10 runs using randomly generated graphs. The X-axis shows the "granularity" of the changes. That is, at one extreme, we apply the changes one at a time, and at the other, we apply all the changes at once and then compute the correctness of the result. Note that the delayed update feature is not used by algorithm `RSJM`, which uses the fact that only one change occurs per test to look for negative cycles. As can be seen, the algorithms that use the adaptive modifications benefit greatly as the batch size is increased, and even among these, the `ABF` algorithm far outperforms the Howard algorithm, because the latter actually performs most of the computation required to compute the maximum cycle-mean of the graph, which is far more than necessary.

Figure 3 shows a plot of what happens when we apply 1000 batches of changes to the graph, but alter the number of changes per batch, so that the total number of changes actually varies from 1000 to 100,000. As expected, `RSJM` takes total time proportional to the number of changes. But the other algorithms take nearly constant time as the batch size varies, which provides the benefit. The reason for the almost constant time seen here is that other bookkeeping operations dominate over the actual computation at this stage. As the batch size increases (asymptotically), we would expect that the adaptive algorithm takes more and more time to operate, finally converging to the same performance as the standard Bellman-Ford algorithm.

As mentioned previously, the adaptive algorithm is better than the incremental algorithm at handling changes in batches. Table 1 shows the relative speedup for different batch sizes on a graph of 1000 nodes and 2000 edges. Although the exact speedup may vary, it is clear that as the number of changes in a batch increases, the benefit of using the adaptive approach is considerable.

Figure 4 illustrates this for a graph with 1000 vertices and 2000 edges. We have plotted this on a log-scale to capture the effect of a large variation in batch size. Because of this, note that the difference in performance between the incremental algorithm and starting from scratch is actually a factor of 3 or so at the beginning, which is considerable. Also, this figure does not show the performance of Howard's algorithm, because as can be seen from figures 2 and 3, the Adaptive Bellman-Ford algorithm considerably outperforms Howard's algorithm in this context.

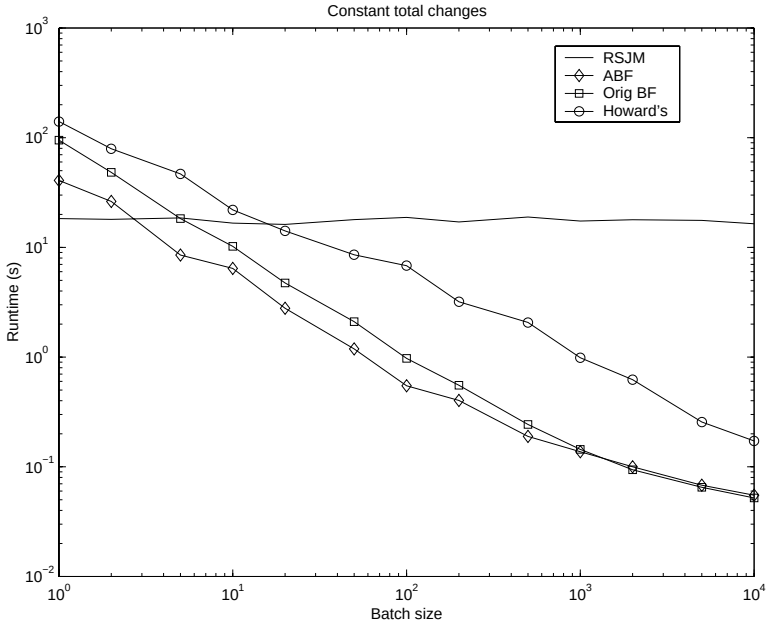


Figure 2: Comparison of algorithms as batch size varies.

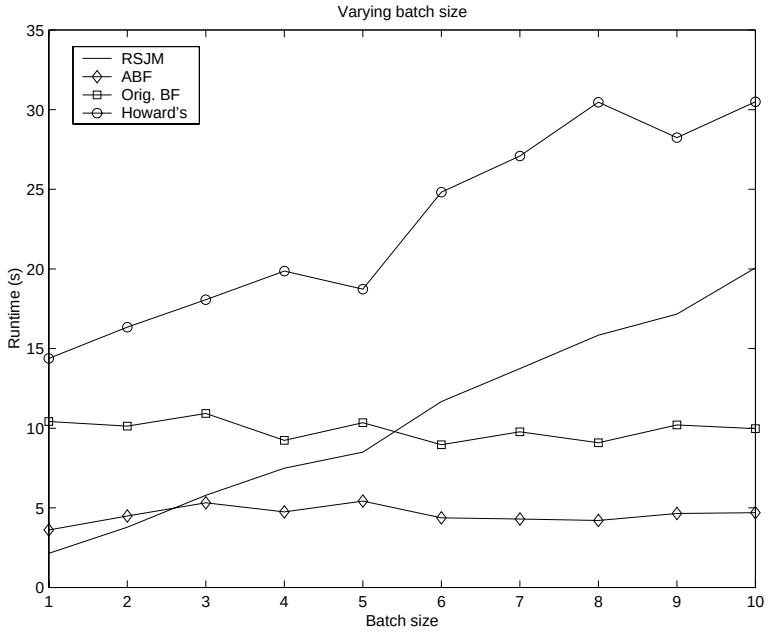


Figure 3: Constant number of iterations at different batch sizes.

Batch size	Speedup (RSJM time/ABF time)
1	0.26×
2	0.49×
5	1.23×
10	2.31×
20	4.44×
50	10.45×
100	18.61×

Table 1: Relative speed of adaptive *vs.* incremental approach for graph of 1000 nodes, 2000 edges

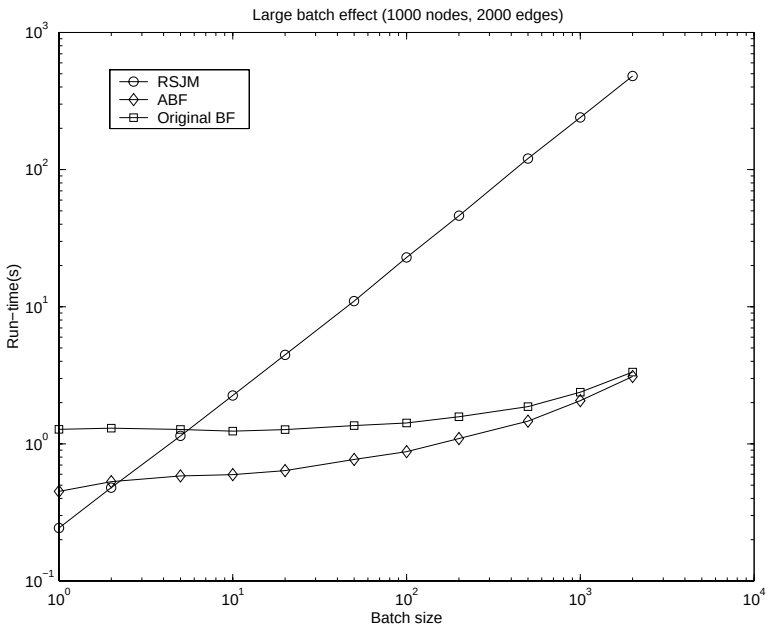


Figure 4: Asymptotic behavior of the algorithms.

An important feature that can be noted from figure 4 is the behavior of the algorithms as the number of changes between updates becomes very large. The `RSJM` algorithm is completely unaffected by this increase, since it has to continue processing changes one at a time. For very large changes, even when we start from scratch, we find that the total time for update starts to increase, because now the time taken to implement the changes itself becomes a factor that dominates overall performance. In between these two extremes, we see that our incremental algorithm provides considerable improvements for small batch sizes, but for large batches of changes, it tends towards the performance of the original Bellman-Ford algorithm for negative cycle detection.

From the figures, we see, as expected, that the `RSJM` algorithm takes time proportional to the total number of changes. Howard's algorithm also appears to take more time when the number of changes increases. Figure 2 allows us to estimate at what batch size each of the other algorithms becomes more efficient than the `RSJM` algorithm. Note that the scale on this figure is also logarithmic.

Another point to note with regard to these experiments is that they represent the relative behavior for graphs with 1,000 vertices and 2,000 edges. These numbers were chosen to obtain reasonable run-times on the experiments. Similar results are obtained for other graph sizes, with a slight trend indicating that the "break-even" point, where our adaptive algorithm starts outperforming the incremental approach, shifts to lower batch-sizes for larger graphs.

5 Applications

In this section, we present two applications that make extensive use of algorithms for negative-cycle detection. In addition, these applications also present situations where we encounter the same graph with slight modifications – either in the edge-weights (maximum cycle-mean computation) or in the actual addition and deletion of a small number of edges (scheduling search techniques). As a result, these provide good examples of the type of applications that would benefit from the adaptive solution to the negative cycle detection problem. As mentioned in the introduction, these problems are central to the high-level synthesis of DSP systems.

5.1 Maximum Cycle Mean computation

The first application we consider is the computation of the Maximum Cycle-Mean (MCM) of a weighted digraph. This is defined as the maximum over all directed cycles of the sum of the arc weights divided by the number of "delay" elements on the arcs. This metric plays an important role in discrete systems and embedded systems [11, 17], since it represents the greatest throughput that can be extracted from the system. Also, as mentioned in [17], there are situations where it may be desirable to recompute this measure several times on closely related graphs, for example for the purpose of design space exploration. As specific examples, [19] proposes an algorithm for dataflow graph partitioning where the repeated computation of the MCM plays a key role, and [4] discusses the utility of frequent MCM computation to synchronization optimization in embedded multiprocessors. Therefore, efficient algorithms for this problem can make it reasonable to consider using such solutions instead of the simpler heuristics that are otherwise necessary. Although several results such as [6, 16] provide polynomial time algorithms for the problem of MCM computation, the first extensive study of algorithmic alternatives for it has been undertaken by Dasdan *et al.* [11]. They concluded that the best existing algorithm in practice for this problem appears to be Howard's algorithm, which, unfortunately, does not have a known polynomial bound on its running time.

To model this application, the edge weights on our graph are obtained from the equation

$$weight(u \rightarrow v) = delay(e) \times P - exec\_time(u),$$

where $weight(e)$ refers to the weight of the edge $e : u \rightarrow v$, $delay(e)$ refers to the number of delay elements (flip-flops) on the edge, $exec\_time(u)$ is the propagation delay of the circuit element that is the source of the vertex, and P is the desired clock period that we are testing the system for. In other words, if the graph

with weights as mentioned above does not have negative cycles, then P is a feasible clock for the system. We can then perform a binary search in order to compute P to any precision we require. This algorithm is attributed to Lawler [18]. Our contribution here is to apply the adaptive negative-cycle detection techniques to this algorithm and analyze the improved algorithm that is obtained as a result.

5.1.1 Experimental setup

For an experimental study, we build on the work by Dasdan and Gupta [11], where the authors have conducted an extensive study of algorithms for this problem. They conclude that Howard's algorithm [8] appears to be the fastest experimentally, even though no theoretical time bounds indicate this. As will be seen, our algorithm performs almost as well as Howard's algorithm on several useful sized graphs, and especially on the circuits of the ISCAS 89/93 benchmarks, where our algorithm typically performs better.

For comparison purposes, we implemented our algorithm in the C programming language, and compared it against the implementation provided by the authors of [8]. Although the authors do not claim their implementation is the fastest possible, it appears to be a very efficient implementation, and we could not find any obvious ways of improving it. As we mentioned in the previous section, the implementation we used incorporates the improvements proposed by Dasdan *et al.* [11]. The experiments were run on a Sun Ultra SPARC-10 (333MHz processor, 128MB memory). This machine would classify as a medium-range workstation under present conditions.

It is clear that the best performance bound that can be placed on the algorithm as it stands is $O(|V||E| \log T)$ where T is the maximum value of P that we examine in the search procedure, and $|V|$ and $|E|$ are respectively the size of the input graph in number of vertices and edges. However, our experiments show that it performs significantly faster than would be expected by this bound.

One point to note is that since we are doing a binary search on T, we are forced to set a limit on the precision to which we compute our answer. This precision in turn depends on the maximum value of the edge-weights, as well as the actual precision desired in the application itself. Since these depend on the application, we have had to choose values for these. We have used a random graph generator that generates integer weights for the edges in the range [0-10,000]. For this range of weights, it could be argued that integer precision would be sufficient. However, since the maximum cycle-mean is a ratio, it is not restricted to integer values. We have therefore conservatively chosen a precision of 0.001 for the binary search (that is, 10^{-7} times the maximum edge-weight). Increasing the precision by a factor of 2 requires one more run of the negative-cycle detection algorithm, which would imply a proportionate increase in the total time taken for computation of the MCM.

With regard to the ISCAS benchmarks, note that there is a slight ambiguity in translating the net-lists into graphs. This arises because a D-type flip-flop can either be treated as a single edge with a delay, with the fanout proceeding from the sink of this edge, or as k separate edges with unit delay emanating from the source vertex. In the former treatment, it makes more sense to talk about the $|D|/|V|$ ratio ($|D|$ being the number of D flip-flops), as opposed to the $|D|/|E|$ ratio that we use in the experiments with random graphs. However, the difference between the two treatments is not significant and can be safely ignored.

We also conducted experiments where we vary the number of edges with delays on them. For this, we need to exercise care, since we may introduce cycles without delays on them, which are fundamentally infeasible and do not have a maximum cycle-mean. To avoid this, we follow the policy of treating edges with delays as "back-edges" in an otherwise acyclic graph [12]. This view is inspired by the structure of circuits, where a delay element usually figures in the feedback portion of the system. Unfortunately, one effect of this is that when we have a low number of delay edges, the resulting graph tends to have an asymmetric structure: it is almost acyclic with only a few edges in the reverse "direction". It is not clear how to get around this problem in a fashion that does not destroy the symmetry of the graph, since this requires solving the *feedback arc set* problem, which is NP-hard [15].

One effect of this is in the way it impacts the performance of the Bellman-Ford algorithm. When the number of edges with delays is small, there are several negative weight edges, which means that the standard Bellman-Ford algorithm spends large amounts of time trying to compute shortest paths initially.

The incremental approach, however, is able to avoid this excess computation for large values of T, which results in its performance being considerably faster when the number of delays is small.

Intuitively, therefore, for the above situation, we would expect our algorithm to perform better. This is because, for the MCM problem, a change in the value of P for which we are testing the system will cause changes in the weights of those edges which have delays on them. If these are fewer, then we would expect that fewer operations would be required overall when we retain information across iterations. This is borne out by the experiments as discussed in the next section.

Our experiments focus more on the kinds of graphs that appear to represent "real" graphs. By this we mean graphs for which the average out-degree of a vertex (number of edges divided by number of vertices), and the relative number of edges with delays on them are similar to those found in real circuits. We have used the ISCAS benchmarks as a good representative sample of real circuits, and we can see that they show remarkable similarity in the parameters we have described: the average out-degree of a vertex is close to and a little less than 2, while an average of about 1/10th or fewer edges have delays on them. An intuitive explanation for the former observation is that most real circuits are usually built up of a collection of simpler systems, which predominantly have small numbers of inputs and outputs. For example, logic gates have typically 2 inputs and 1 output, as do elements such as adders and multipliers. More complex elements like multiplexers and encoders are relatively rare, and even their effect is somewhat offset by single-input single-output units like `NOT` gates and filters.

5.1.2 Experimental results

We now present the results of the experiments on random graphs with different parameters of the graph being varied.

We first consider the behavior of the algorithms for random graphs consisting of 10,000 vertices and 20,000 edges, when the "feedback-edge ratio" (ratio of edges with non-zero delay to total number of edges) is varied from 0 to 1 in increments of 0.1. The resulting plot is shown in Fig. 5. As discussed in the previous section, for small values of this ratio, the graph is nearly acyclic, and almost all edges have negative weights. As a result, the normal Bellman-Ford algorithm performs a large number of computations that increase it's running time. The ABF-based algorithm is able to avoid this overhead due to its property of retaining information across runs, and so it performs significantly better for small values of the feedback edge ratio. The ABF based algorithm and Howard's algorithm perform almost identically in this experiment. The points on the plot represent an average over 10 random graphs each.

Figure 6 shows the effect of varying the number of vertices. The average degree of the graph is kept constant, so that there is an average of 2 edges per vertex, and the feedback edge ratio is kept constant at 1 (all edges have delays). The reason for the choice of average degree was explained in section 5.1.1. Figure 7 shows the same experiment, but this time with a feedback edge ratio of 0.1. We have limited the displayed portion of the Y-axis since the values for the MCM computation using the original Bellman-Ford routine rise as high as 10 times that of the others and drowns them out otherwise.

These plots reveal an interesting point: as the size of the graph increases, Howard's algorithm performs less well than the MCM computation using the Adaptive Bellman-Ford algorithm. This indicates that for real circuits, the ABF-based algorithm may actually be a better choice than even Howard's algorithm. This is borne out by the results of the ISCAS benchmarks.

Figure 8 and figure 9 are a study of what happens as the edge-density of the graph is varied: for this, we have kept the number of edges constant at 20,000, and the number of vertices varies from 1,000 to 17,500. This means a variation from an edge-density (ratio of number of edges to number of vertices) of 1.15 to 20. In both these figures, we see that the MCM computation using ABF performs especially well at low densities (sparse graphs), where it does considerably better than Howard's algorithm and the normal MCM computation using ordinary negative cycle detection. In addition, the point where the ABF-based algorithm starts performing better appears to be at around an edge-density of 2, which is also seen in figure 5.

We note the following features from the experiments:

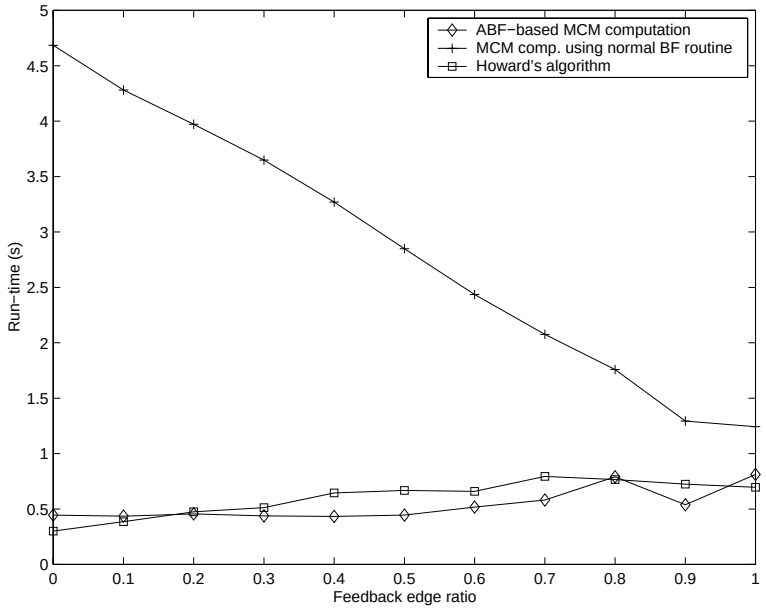


Figure 5: Comparison of algorithms for 10,000 vertices, 20,000 edges: the number of feedback edges (with delays) is varied as a proportion of the total number of edges.

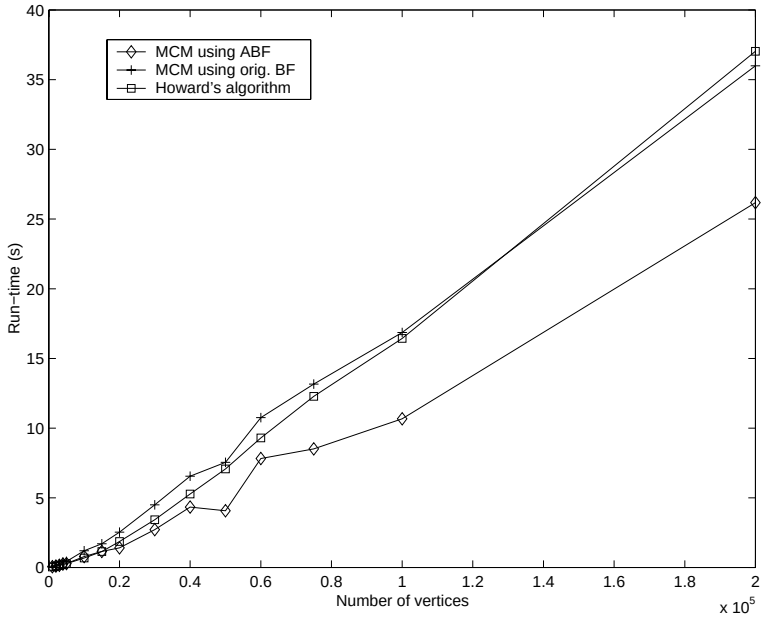


Figure 6: Performance of the algorithms as graph size varies : all edges have delays (feedback edges) and number of edges = twice number of vertices.

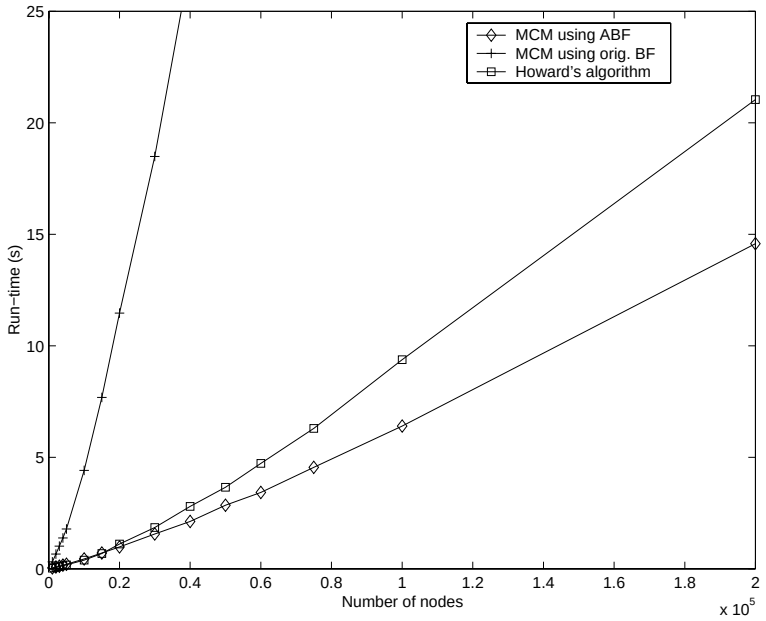


Figure 7: Performance of the algorithms as graph size varies: proportion of edges with delays = 0.1 and number of edges = twice number of vertices (Y-axis limited to show detail).

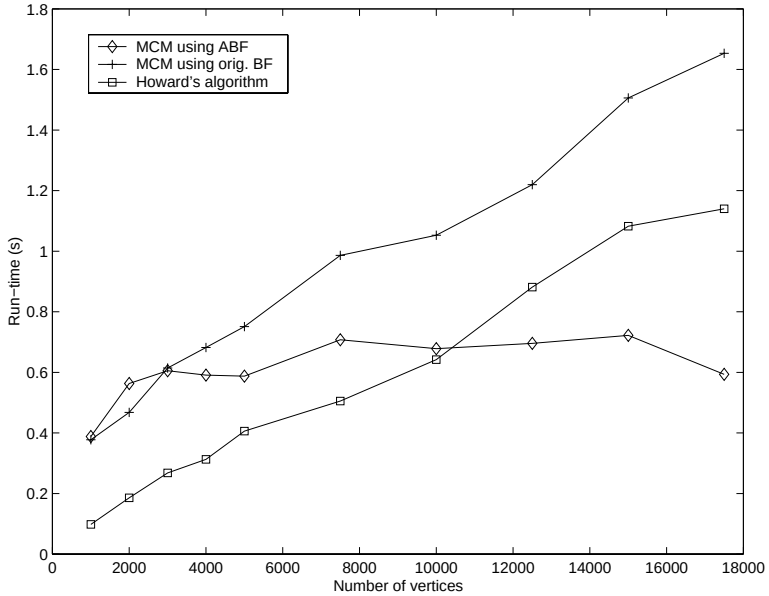


Figure 8: Performance of the algorithms as graph edge density varies: all edges have delays (feedback edges) and number of edges = 20,000.

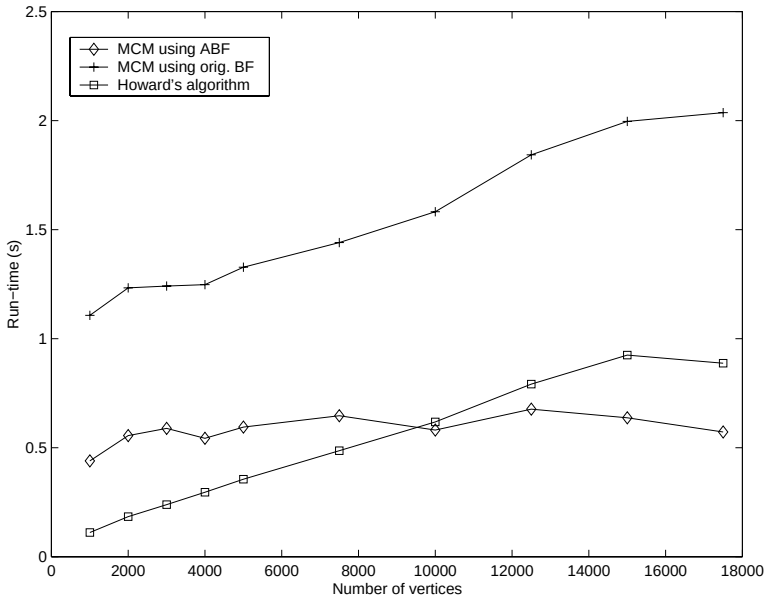


Figure 9: Performance of the algorithms as graph size varies: proportion of edges with delays = 0.1 and number of edges = 20,000.

- If all edges have unit delay, the MCM algorithm that uses our adaptive negative cycle detection provides some benefit, but less than in the case where few edges have delays.
- When we vary the number of feedback edges (edges with delays), the benefit of the modifications becomes very considerable at low feedback ratios, doing better than even Howard's algorithm for low edge densities.
- In the ISCAS benchmarks, we can see that all of the circuits have $|E|/|V| < 2$, and $|D|/|V| < 0.1$, ($|D|$ is number of flip-flops, $|V|$ is total number of circuit elements, and $|E|$ is number of edges). In this range of parameters, our algorithm performs very well, even better than Howard's algorithm in several cases (also see Table 2 for our results on the ISCAS benchmarks).

Table 2 shows the results obtained when we used the different algorithms to compute MCMs for the circuits from the ISCAS 89/93 benchmark set. One point to note here is that the ISCAS circuits are not true HLS benchmarks: they were originally designed with logic circuits in mind, and as such, the normal assumption would be that all registers (flip-flops) in the system are triggered by the same clock. In order to use them for our testing, however, we have relaxed this assumption and allowed each flip-flop to be triggered on any phase: in particular, the phases that are computed by the MCM computation algorithm are such that the overall system speed is maximized. These benchmark circuits are still very important in the area of HLS, because real DSP circuits also show similar structure (sparseness and density of delay elements), and an important observation we can make from the experiments is that the structure of the graph is very relevant to the performance of the various algorithms in the MCM computation.

As can be seen in the table, Lawler's algorithm does reasonably well at computing the MCM. However, when we use the adaptive negative cycle detection in place of the normal negative cycle detection technique, there is an increase in speed by a factor of 5 to 10 in most cases. This increase in speed is in fact sufficient to make Lawler's algorithm with this implementation up to twice as fast as Howard's algorithm, which was otherwise considered the fastest algorithm in practice for this problem.

Benchmark	$\frac{\lvert E \rvert}{\lvert V \rvert}$	$\frac{\lvert D \rvert}{\lvert V \rvert}$	orig.BF MCM	ABF MCM	Howard's algo.
s38417	1.416	0.069	2.71	0.29	0.66
s38584	1.665	0.069	2.66	0.63	0.59
s35932	1.701	0.097	1.79	0.37	0.09
s15850	1.380	0.057	1.47	0.18	0.36
s13207	1.382	0.077	0.73	0.12	0.35
s9234	1.408	0.039	0.57	0.06	0.11
s6669	1.657	0.070	0.74	0.07	0.04
s4863	1.688	0.042	0.27	0.04	0.03
s3330	1.541	0.067	0.11	0.02	0.01
s1423	1.662	0.099	0.07	0.01	0.01

Table 2: Run-time for MCM computation for the 6 largest ISCAS 89/93 benchmarks.

5.2 Search Techniques for Scheduling

In this section, we demonstrate another application of our technique: efficient searching of schedules for iterative dataflow graphs (I-DFGs). The basic idea is that for scheduling an IDFG, we need to (a) assign vertices to processors and (b) assign relative positions to the vertices within each processor (for resource sharing). Once these two aspects are done, the schedule for a given throughput constraint is determined by finding a feasible solution to the constraint equations, which we do using the ABF algorithm. This idea that the ordering can be directly used to compute schedule times has been used previously, for example see [27]. Since the search process involves repeatedly checking the feasibility of many similar constraint systems, the advantages of the adaptive negative cycle detection come into play.

The approach we have taken for the schedule search is:

- Start with each vertex on its own processor, find a feasible solution on the fastest possible processor.
- Examine each vertex in turn, and try to find a place for it on another processor (resource sharing). In doing so, we are making a small number of changes to the constraint system, and need to recompute a feasible solution.
- In choosing the new position, choose one that has minimum power (or area, or whatever cost we want to optimize).
- Additional "moves" that can be made include inserting a new processor type and moving as many vertices onto it as possible, moving vertices in groups from one processor to another, etc.
- The technique also lends itself very well to application in schemes using evolutionary improvement [3].
- In the present implementation, to choose among various equivalent implementations at a given stage, we use a weight based on giving greater importance to implementations that result in lower overall slack on the cycles in the system. ("Slack" of a cycle here refers to the difference between the total delay afforded by the registers (number of delay elements in the cycle times the clock period) and the sum of the execution times of the vertices on the cycle, and is useful since a large slack could be taken as an indication of under-utilization of resources.)

Each such "move" or modification that we make to the graph can be treated as a set of edge-changes in the precedence/processor constraint graph, and a feasible schedule would be found if the system does not have negative cycles. In addition, the $dist(v)$ values that are obtained from applying the algorithm directly give us the starting times that will meet the schedule requirements.

We have applied this technique to attack the multiple-voltage scheduling problem addressed in [26]. The problem here is to find a schedule for the given DFG that minimizes the overall power consumption, subject

Example	Res. (5V+, 3.3V+,5V*)	T	Power saved	
			S and R	ABF
5th-order ellip.filt.	{2, 2, 2}	25	31.54%	34.86%
	{2, 1, 2}	25	18.26%	16.60%
	{2, 2, 2}	22	23.24%	26.56%
	{2, 1, 2}	21	13.28%	14.94%
FIR filt.	{1, 2, 1}	15	29.45%	×
	{1, 2, 2}	15	–	34.35%
	{1, 2, 1}	16	–	36.81%
	{1, 2, 2}	10	17.18%	24.54%

Table 3: Comparison between ABF-based search and algorithm of Sarrafzadeh and Raje (× → failed to schedule, – → not available)

to fixed constraints on the iteration period bound, and on the total number of resources available. For this example, we consider three resources: adders that operate at 5V, adders that operate at 3.3V, and multipliers that operate at 5V. For the elliptic filter, multipliers operate in 2 time units, while for the FIR filter, they operate in 1 time unit. The 5V adders operate in 1 time unit, while the 3.3V adders operate in 2 time units always. It is clear that the power savings are obtained through scheduling as many adders as possible on 3.3V adders instead of 5V adders. We have used only the basic resource types mentioned in the table to compare our results against those in [26]. However, there is no inherent limit imposed by the algorithm itself on the number of different kinds of resources that we can consider.

In tackling this problem, we have used only the most basic method, namely moving vertices onto another existing processor. Already, the results match and even outperform that obtained in [26]. In addition, the method has the benefit that it can handle any number of voltages/processors, and can also easily be extended to other problems, such as homogeneous-processor scheduling [12]. Table 3 shows the power-savings that were obtained using this technique. "S and R power saving" indicates the power savings (assuming 25 units for 5V devices and 10.89 units for 3.3V devices) obtained by [26], while "ABF power savings" refers to the results obtained using our algorithm (where the ABF algorithm is used to test the feasibility of the system after each "move" as per the definition above). T is the overall timing constraint (the maximum iteration period bound that we are aiming for).

Table 3 shows some interesting features: the iterative improvement based on the ABF algorithm (column marked "ABF") produced results with significantly higher power savings than the results presented in [26]. One important reason contributing to this could be that the iterative improvement algorithm makes full use of the iterative nature of the graphs, and produces schedules that make good use of the available inter-iteration parallelism. On the other hand, we find that for one of the configurations, the ABF-based algorithm is not able to find any valid schedule. This is because the simple nature of the algorithm occasionally results in it getting stuck in local minima, with the result that it is unable to find a valid schedule even when one exists.

Several variations on this theme are possible: the search scheme could be used for other criteria such as the case where the architecture needs to be chosen (not fixed in advance), and modifications such as small amounts of randomization could be used to prevent the algorithm from getting stuck in local minima. This flexibility combined with the speed improvements afforded by the improved adaptive negative cycle detection can allow this method to form the core of a large class of scheduling techniques.

6 Conclusions

The problem of negative cycle detection is considered in the context of HLS for DSP systems. It was shown that important problems such as performance analysis and design space exploration often result in the construction of "dynamic" graphs, where it is necessary to repeatedly perform negative cycle detection on

variants of the original graph.

We have introduced an adaptive approach (the ABF algorithm) to negative cycle detection in dynamically changing graphs. Specifically, we have developed an enhancement to Tarjan's algorithm for detecting negative cycles in static graphs. This enhancement yields a powerful algorithm for dynamic graphs that outperforms previously available methods for addressing the scenario where multiple changes are made to the graph between updates. Our technique explicitly addresses the common, practical scenario in which negative cycle detection must be periodically performed after intervals in which a small number of changes are made to the graph. We have shown by experiments that for reasonable sized graphs (10,000 vertices and 20,000 edges) our algorithm outperforms the incremental algorithm (one change processed at a time) described in [24] even for changes made in groups of as little as 4-5 at a time.

As our original interest in the negative cycle detection problem arose from its application to the problems described above in HLS, we have implemented some schemes that make use of the adaptive approach to solve those problems. We have shown how our adaptive approach to negative cycle detection can be exploited to compute the maximum cycle mean of a weighted digraph, which is a relevant metric for determining the throughput of DSP system implementations. We have compared our ABF technique, and ABF-based MCM computation technique against the best known related work in the literature, and have observed favorable performance. Specifically, the new technique provides better performance than Howard's algorithm for sparse graphs with relatively few edges that have delays.

Since computing power is cheaply available now, it is increasingly worthwhile to employ extensive search techniques for solving NP-hard analysis and design problems such as scheduling. The availability of an efficient adaptive negative cycle detection algorithm can make this process much more efficient in many application contexts. We have demonstrated this concretely by employing our ABF algorithm within the framework of a search strategy for multiple voltage scheduling.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*. Upper Saddle River, NJ, USA: Prentice Hall, 1993.

[2] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck, "Incremental evaluation of computational circuits," in *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, pp. 32–42, 1990.

[3] T. Bäck, U. Hammel, and H.-P. Schwefel, "Evolutionary computation: Comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 3–17, Apr. 1997.

[4] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Resynchronization for multiprocessor DSP systems," *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, vol. 47, pp. 1597–1609, November 2000.

[5] N. Chandrachoodan, S. S. Bhattacharyya, and K. J. R. Liu, "Adaptive negative cycle detection in dynamic graphs," in *Proceedings of International Symposium on Circuits and Systems (ISCAS 2001)*, vol. V, (Sydney, Australia), pp. 163–166, May 2001.

[6] D. Y. Chao and D. T. Wang, "Iteration bounds of single rate dataflow graphs for concurrent processing," *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, vol. 40, pp. 629–634, Sep 1993.

[7] B. Cherkassky and A. V. Goldberg, "Negative cycle detection algorithms," Tech. Rep. tr-96-029, NEC Research Institute, Inc., March 1996.

[8] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. McGettrick, and J.-P. Quadrat, "Numerical computation of spectral elements in max-plus algebra," in *Proc. IFAC Conf. on Syst. Structure and Control*, 1998.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[10] A. Dasdan, S. S. Irani, and R. K. Gupta, "An experimental study of minimum mean cycle algorithms," Tech. Rep. UCI-ICS #98-32, Univ. of California Irvine, 1998.

[11] A. Dasdan, S. S. Irani, and R. K. Gupta, "Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems," in *36th Design Automation Conference*, pp. 37–42, ACM/IEEE, 1999.

[12] S. M. H. de Groot, S. H. Gerez, and O. E. Herrmann, "Range-chart-guided iterative data-flow graph scheduling," *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, vol. 39, pp. 351–364, May 1992.

[13] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone, "Experimental analysis of dynamic algorithms for single source shortest paths problem," in *Proc. Workshop on Algorithm Engineering (WAE '97)*, 1997.

[14] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic shortest paths and negative cycle detection on digraphs with arbitrary arc weights," in *ESA98*, vol. 1461 of *Lecture Notes in Computer Science*, (Venice, Italy), pp. 320–331, Springer, August 1998.

[15] M. R. Garey and D. S. Johnson, *Computers and Intractability - A guide to the theory of NP-completeness*. W.H.Freeman and Company, NY, USA, 1979.

[16] S. H. Gerez, S. M. H. de Groot, and O. E. Hermann, "A polynomial time algorithm for computation of the iteration period bound in recursive dataflow graphs," *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, vol. 39, pp. 49–52, Jan 1992.

[17] K. Ito and K. K. Parhi, "Determining the minimum iteration period of an algorithm," *Journal of VLSI Signal Processing*, vol. 11, pp. 229–244, 1995.

[18] E. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rhinehart and Winston, 1976.

[19] L.-T. Liu, M. Shih, J. Lillis, and C.-K. Cheng, "Data-flow partitioning with clock period and latency constraints," *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, vol. 44, Mar 1997.

[20] K. Mehlhorn and S. Näher, "`LEDA`: A platform for combinatorial and geometric computing," *Communications of the ACM*, vol. 38, no. 1, pp. 96–102, 1995.

[21] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Transactions on Computers*, vol. 40, pp. 178–195, Feb 1991.

[22] G. Ramalingam, *Bounded Incremental Computation*. PhD thesis, University of Wisconsin, Madison, August 1993. Revised version published by Springer Verlag (1996) as Lecture Notes in Computer Science 1089.

[23] G. Ramalingam and T. Reps, "An incremental algorithm for a generalization of the shortest-paths problem," *Journal of Algorithms*, vol. 21, pp. 267–305, 1996.

[24] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller, "Solving systems of difference constraints incrementally," *Algorithmica*, vol. 23, pp. 261–275, 1999.

[25] R. Reiter, "Scheduling parallel computations," *Journal of the ACM*, vol. 15, pp. 590–599, Oct 1968.

[26] M. Sarrafzadeh and S. Raje, "Scheduling with multiple voltages under resource constraints," in *Proc. ISCAS 99*, 1999.

[27] D. J. Wang and Y. H. Hu, "Fully static multiprocessor array realizability criteria for real-time recurrent DSP applications," *IEEE Transactions on Signal Processing*, vol. 42, pp. 1288–1292, May 1994.