Acceleration of Algebraic Recurrences on Processors with Instruction Level Parallelism

Michael Schlansker, Vinod Kathail
Computer Systems Laboratory

HPL-93-55
July, 1993

HPL-93-55: Acceleration of Algebraic Recurrences on Processors with Instruction Level Parallelism

Acceleration of Algebraic Recurrences on Processors with Instruction Level Parallelism

Michael Schlansker, Vinod Kathail
Computer Systems Laboratory
HPL-93-55
July, 1993

email: schlansk@hplabs.hpl.hp.com
kathail@hplabs.hpl.hp.com

recurrences,
parallelism,
software-pipeline,
loop optimization,
height reduction,
back-substitution,
blocked back-substitution

This report describes parallelization techniques for accelerating a broad class of recurrences on processors with instruction level parallelism. We introduce a new technique, called blocked back-substitution, which has lower operation count and higher performance than previous methods. The blocked back-substitution technique requires unrolling and non-symmetric optimization of innermost loop iterations. We present metrics to characterize the performance of software-pipelined loops and compare these metrics for a range of height reduction techniques and processor architectures.

To be published in the *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 12-14, 1993

1 Introduction

Architectures with instruction level parallelism such as VLIW and superscalar processors provide parallelism in the form of a limited number of pipelined functional units. For these architectures, recurrence height reduction techniques provide significant speedups when they are properly applied. This paper introduces a new technique, called **blocked back-substitution**, which preserves the height reduction benefits observed with symmetric back-substitution techniques but has substantially lower operation count. We analyze the impact of both the latency and the number of operations used to evaluate a recurrence on the performance of software-pipelined loops. We present speedup results on a variety of example architectures to demonstrate the utility of the techniques presented in this paper.

Recurrences appear in many forms within application programs and typically result in a performance bottleneck when executed on a processor with instruction level parallelism. A chain of operations connected through flow dependence limits program performance. Most recurrences are either first or second order where efficient acceleration techniques are best demonstrated. We believe that a broad class of recurrences can be efficiently accelerated, and we demonstrate the acceleration of one important class of recurrences within this paper.

The height reduction of arithmetic recurrences has been extensively studied in the literature; see, for example, [1-5]. The techniques used in these papers introduce computational redundancy in order to reduce the solution time of a recurrence on a parallel computer but do not carefully optimize the solution of recurrences for processors with limited instruction level parallelism. A frequently discussed solution is cyclic reduction. When cyclic reduction is applied, the operation count is order n log n leading to excessive operation count for large problem sizes.

The partition method was introduced by Chen et. al. [6] and studied further by H. H. Wang [7] and H. A. Van Der Vorst et. al. [8]. The partition method is amenable to parallelization and results in lower operation count (approximately 5n) when solving a first order linear recurrence. In this technique, a simple recurrence loop with trip count n is transformed into multiple nested loops with smaller innermost trip count (approximately square root of n). When a recurrence loop has mixed recurrences of different type, a loop exit, or other complex code structures, the transformation of a single innermost loop to a complex multiple loop configuration may be very difficult or impossible. The dismantling of a single loop into multiple nested loops also increases vector startup penalties and requires more load store instructions than optimal methods.

Work by Tanaka et. al. [9] introduces a height reduction technique similar to blocked back-substitution. The technique is used to accelerate first order linear recurrences on a vector computer that has been augmented with a hardware floating point recurrence solution instruction. Our interest is in more flexible processor architectures with significant instruction level parallelism. We describe a broader family of height reduction techniques and use these techniques to efficiently process algebraic recurrences of first and higher order without the introduction of any specialized instructions.

Work by Callahan [10] introduces a fairly general class of recurrences called "bounded recurrences" . A primary objective of Callahan is to establish a notational framework providing for the general description of recurrence acceleration techniques. The work focuses primarily on the acceleration of recurrences for multiprocessors. We treat specifically the acceleration of recurrences on uniprocessors with instruction level parallelism.

Two loop acceleration techniques, are popular in the literature. In trace (or superblock) scheduling [11-13], loops are unrolled into a trace. Iterations within a single trace are overlapped, but successive traces are not. The schedule length of the trace is greater than or equal to a maximal height required by dependence over all pairs of operations in the trace. Algebraic height reductions must accommodate the distance between all operations in the trace. In software pipelining [14, 15], successive loop iterations are identical. Height reduction must minimize the relative height between each iteration and identical prior iterations In software pipelines, successive iterations overlap, and only operations on time critical dependence paths are height reduced. Block back-substitution combines these techniques and schedules unrolled loops of conventional instructions in software pipelines.

Blocked back-substitution is a general algebraic height reduction technique for recurrences providing lower operation count and higher performance than prior techniques. Blocked back-substitution is also applicable to non-linear recurrences when the primitive operations in which the recurrence is expressed have the associative and distributive properties. The technique is especially advantageous for machines with a large degree of parallelism obtained through either pipelining or a large number of function units. The use of blocked back-substitution accelerates an innermost loop recurrence using a height reduced innermost loop. The preservation of the original innermost loop facilitates the scheduling of other innermost loop code in a common software pipeline with height reduced recurrence code. The fused loop has reduced vector startup penalty when compared to solutions requiring multiple fragmented loops.

The report is organized as follows. The rest of this section provides an overview of asymmetric height reduction and introduces notation and terminology used in the rest of the report. Section 2 provides an overview of recurrence height reduction schema useful within software pipelines. Height reduction schema such as symmetric back-substitution and blocked back-substitution are defined for later comparison. Section 3 specializes symmetric and blocked back-substitution to the case of a first order multiply-add recurrence. Critical path lengths and operation counts allow direct performance comparison between the two approaches. Section 4 generalizes this work by providing a common technique to accelerate multiply-add recurrences of any order. Section 5 provides a detailed comparison of the performance of these techniques. Section 6 contains concluding remarks.

1.1 Overview of Asymmetric Height Reduction

Many of the height reduction techniques found in the literature use a balanced approach in the sense that they try to minimize the length of the path from each of the inputs to the output of an expression. The balanced approach usually requires excessive redundant work, but may offer excellent speedup when the critical path length through the computation is measured. When compiling for a small number of processing elements, balanced approaches may be too computationally expensive especially when height reductions of deeply nested recurrences are required.

In this work, we have given considerable attention to minimizing the redundant work required to execute a computation. This is accomplished by evaluating expressions using asymmetric height reductions. Consider the following example:

$$s_o = c_1 + x_1(c_2 + x_2(c_3 + x_3(c_4 + x_4(c_5 + x_5(c_6 + x_6(c_7 + x_7(c_8 + x_8(s_i)))))))$$

The evaluation of this expression is sequential requiring eight multiply and eight add operations. A height reduction goal is needed to characterize the successful parallelization of this expression.

One goal is to jointly minimize the height from each of the input c_i's and s_i to the output s_o In important cases, this goal requires excessive redundant computation and yields no additional performance benefit. A more easily met goal minimizes only a critical path (e.g. from s_i to s_o). Here, we assume that all of the c_i's are available well in advance of s_i and the evaluation latency from each c_i to s_o is not important. We use the associative and distributive properties to derive the following expression which has a single multiplication and addition on the path from s_i to s_o and requires seven additional multiplies as compared to the original expression.

$$s_o = (x_1x_2x_3x_4x_5x_6x_7x_8)s_i + (c_1 + x_1(c_2 + x_2(c_3 + x_3(c_4 + x_4(c_5 + x_5(c_6 + x_6(c_7 + x_7(c_8)))))))$$

Balanced height reductions (those which jointly expedite the paths from all of the c_i's to s_o) result in substantially increased operation count yet still require at least a single multiplication and addition on the path from s_i to s_o. Such algebraic transformations have been extensively explored in techniques for the acceleration of carry look-ahead.

Another very important asymmetric evaluation principle explored within this paper is the use of blocked height reductions in the acceleration of recurrences. Here, we expedite the evaluation of a sequence of values originally calculated by repeatedly executing a recurrence expression in a loop. When a recurrence is blocked, a simple loop with recurrence is unrolled by a blocking factor to create an asymmetric loop body. Height reduction is applied to some of the iterations within the loop body, while others are evaluated using the non-height reduced original code.

In this approach, a sequence of recurring values is separated into two categories: The timely evaluation of some of the values is expedited through height reduction with requisite computational redundancy. The evaluation of other values is not time critical and is performed using the original expression, which involves no redundant computation. This asymmetric treatment combines the height reduction benefits yielded by redundant expression evaluation with reduced operation count resulting from using redundancy as sparingly as possible.

The blocked back-substitution introduced in this paper uses associative and distributive properties to reduce the height of expressions, and it uses common sub-expression elimination to minimize the number of operations executed. In many cases, the blocked back-substitution technique provides unlimited parallelism with a corresponding increase in redundant operation count which is bounded by a constant multiplier.

1.2 Loop Recurrence Notation

This notation used to describe the flow of values between loop iterations is taken from a paper by Rau [16]. In this paper, we use the concept of the **expanded virtual register (EVR)** which is a linearly ordered set of virtual registers with a special remap operation. Note that the use of the EVR is not restricted to processors with hardware support for rotating registers. Compilation techniques using virtual rotating registers are suitable for conventional register files but require the replication of loop program text as described in a paper by Rau et. al. [17].

A scalar value s (usually calculated repeatedly within the body of a loop) can be referenced using the notation s[i]. Each time a **remap(s)** is executed, all values in the sequence shift "upward" so that each value s[i] prior to the execution of a remap is renamed as s[i+1] after the remap. Thus,

the value s (identically equal to s[0]) after the execution of a single remap is renamed as s[1]. This allows multiple values from a sequence of scalar assignments to remain alive without the automatic overwrite of a value when the next member of the sequence is computed. This notation is particularly useful for the description of value usage in higher order recurrences.

We must also describe interface specifications between the code for a loop and the code that precedes or follows the loop code. For a variable s calculated in a recurrence within a loop, we introduce s_{in} and s_{out} to precisely describe loop interfaces. Out of loop code prior to the loop establishes an initial value for the s which we will call s_{in} . After loop execution has completed, the repeated evaluation of the loop body establishes a final value for the recurring value s known as s_{out}, which is referenced by out of loop code subsequent to loop execution.

1.3 Loop Performance Metrics

We use four performance metrics to help describe the advantages and disadvantages of specific recurrence acceleration methods. We define a simple metric **Ops/iter** which counts the average number of operations required to execute a single iteration of the recurrence loop. One can compare the number of operations per iteration prior to applying a height reduction technique against the number of operations per iteration after height reduction to measure redundant work introduced for the purposes of acceleration. The Ops/iter measure assumes that operations of all types are of equal cost. When a loop body has been unrolled and iterations are not treated symmetrically, the Ops/iter measure averages the number of operations required per iteration across the unrolled loop block.

When a specific processor and corresponding code for a software pipeline has been selected, we can use Ops/iter to calculate **ResMII**. ResMII and the following two metrics have been described in Dehnert et. al. [18]. ResMII establishes a resource bound on the rate of execution of loop iterations within software pipelines. In the traditional definition of ResMII, if a processor has u identical function units each capable of executing any operation, then:

$$\mathrm{ResMII} = \left\lceil \frac{\mathrm{Ops / iter}}{u} \right\rceil .$$

This expresses the constraint that if a schedule is developed which saturates a critical resource, the number of cycles per iteration cannot be less than the ResMII. The formula uses a ceiling function to constrain identical loop iterations in a software pipeline to take an integral number of cycles. We present techniques which use loop unrolling where the schedule for successive iterations is not identical and this constraint is pessimistic. We modify the ResMII calculation to provide a smaller (truly lower) bound for the unrolled loop case:

$$\mathrm{ResMII} = \frac{\mathrm{Ops / iter}}{u} .$$

Here, ResMII provides a lower (possibly non-integral) bound on the average number of iterations required per iteration across the unrolled block.

The **RecMII** of a loop describes a bound on the rate at which a software pipeline is executed based on a critical path measurement through operations within the body of the loop. Given an expression with recurrence evaluated in the body of a loop and specific hardware latencies, the RecMII is calculated as a worst case path length computation over all circuits in the program

graph. Each circuit is normalized by the number of iterations spanned by the circuit to correctly calculate its impact on throughput. Once again, we have adapted RecMII to the unrolled loop case. We calculate the RecMII for the unrolled loop body and divide by the number of iterations unrolled. This provides a potentially non-integral lower bound on the number of cycles per source iteration in the case of unrolled loops.

The **MII** of a loop is a performance bound that takes both resource and critical path constraints into account. It is simply the maximum of ResMII and RecMII. Once again, we use a potentially non-integral form for MII derived through this maximum to better model unrolled loops.

2 Recurrence Acceleration Schemas

In this section, we illustrate three loop acceleration methods using the same program example. The example program consists of a simple associative reduction. It is commonly the case that a sequence of terms within a loop are collected into a single result using an associative operation. This action is termed associative reduction. Associative reduction can be separated into two broad classes:

1. **reduction to scalar** in which a sequence of terms calculated within the loop is reduced to a single scalar which is used after loop execution, and
2. **reduction to vector** in which terms from all iterations prior to an iteration are associatively accumulated and either stored to memory or otherwise used within the loop necessitating the correct calculation of the entire sequence of values.

These two cases require distinct treatment. In the following examples, we use the term "addition" and the operation + to represent an arbitrary associative operation and we use ℓ to represent its latency in processor cycles.

2.1 Interleaved Reduction

We first describe an interleaved reduction method which is useful in the associative reduction of a number of terms to a scalar. It has been implemented within the Cydra 5 compiler (see [18] in which the method is called "riffled" reduction) and within the Trace compiler [12]. Consider the following pseudo code:

```
s[1] = s_in
do i=1,n
s = s[1] + a_i
remap(s)
enddo
s_out = s[1]
```

Original code for a reduction to scalar loop

In this example, we assume that an initial scalar value live in to the body of the loop s_{in} is copied into s[1] and, the final value s_{out} is copied from s[1] after loop execution completes. The performance metrics for this loop are as follows: Ops / iter = 1 and RecMII = ℓ. The

performance metrics indicate that only a single addition is required per cycle but no more than one iteration can be retired every ℓ cycles.

Interleaved reduction increases the degree of parallelism without redundant work and requires no replication of the loop body and thus, is an excellent schema for the associative reduction to scalar. The single dependence chain of operations defined within the original loop is split into b independent chains thus effecting a b-fold height reduction. The use of interleaved reduction to scalar is illustrated in the program below.

```
s[1] = s_in,   s[2] = 0,   s[3] = 0, ···  s[b] = 0
do i=1,n
s = s[b] + a_i
remap(s)
enddo
s_out = s[1] + s[2] + ··· + s[b]
```

Interleaved reduction

Interleaved reduction results in the following loop performance metrics: $\text{Ops} / \text{iter} = 1$ and $\text{RecMII} = \frac{\ell}{b}$. Interleaved reduction is not suitable for associative reduction to vector because it correctly calculates only the final term in the original reduction series. Interleaved reduction is also not useful when treating more complex recurrences such as first order and higher order linear recurrences described below. We provide no further discussion of interleaved reduction, rather we will focus on the next two techniques.

2.2 Symmetric Back-substitution

Symmetric back-substitution has also been implemented within the Cydra 5 compiler. The technique is more general than interleaved reduction because it handles more complex recurrences and correctly treats the reduction to vector case. However, we will show that symmetric back-substitution is frequently not the best approach. We have modified the original code shown above to illustrate the reduction to vector case. Here, the entire stream of values of s are used (stored into the array x(i)) within the body of the loop. Again the performance metrics for the unmodified loop are expressed as: $\text{Ops} / \text{iter} = 1$ and $\text{RecMII} = \ell$ indicating that only a single addition is required per cycle but no more than one iteration can be retired every ℓ cycles where ℓ represents the addition latency.

```
s[1] = s_in
do i=1,n
s = s[1] + a_i
x(i) = s
remap(s)
enddo
```

Original code for a reduction to vector loop

The following presentation focuses on the rapid evaluation of the recurring value sequence and not the means by which the values are actually used. We assume that either or both the entire value sequence or the final live out value within a loop are potentially used but, we at times omit these uses. Schema presented below must be optimized to accommodate whether such uses exist.

Back-substitution is achieved by substituting the expression for s computed within a previous iteration ($s[1] = s[2] + a_{i-1}$) into the expression for s computed within the current iteration. Thus we get $s = (s[2] + a_{i-1}) + a_i$. The process may be repeated to any desired depth b of back-substitution. The associative property of addition is used to asymmetrically height reduce the result (e.g. expedite the s[2] to s[0] path for b=2). This yields the expression $s = s[2] + (a_{i-1} + a_i)$. The technique produces each term in the sequence of values faithfully and thus can be used for both the reduction to scalar and reduction to vector cases. The computational work increases linearly in b as we expose additional parallelism. This yields inefficient solutions when highly parallelized forms of the back-substituted code are used. Code for symmetric back-substitution is illustrated below:

```
s[1] = s_in
do i=1,b-1      /*treat b-1 iterations conventionally*/
s = s[1] + a_i
remap(s)
enddo

do i=b,n        /*b fold back-substitution of remaining
iterations*/
s = s[b] + (a_i + a_{i-1} + a_{i-2} + ... + a_{i-b-1})
remap(s)
enddo
```

Code after symmetric back-substitution

The second loop depicts code which has been parallelized through back-substitution. The back-substitution process has changed the loop from a first order recurrence to a b^{th} order recurrence. If one enters the second loop directly this higher order recurrence uses the value s[b] which is not properly defined. The higher order recurrence is correctly initialized by executing b-1 conventional iterations within the first unmodified loop. The remaining n-(b-1) iterations are executed within the second higher order recurrence loop.

Loop performance metrics for the second loop are: $Ops / iter = b$ and $RecMII = \frac{\ell}{b}$. When one applies a b fold height reduction, a resulting b-fold increase in work is required. The linear growth in the computational requirements for symmetric back-substitution motivates the blocked back-substitution approach presented below.

We note that in the case of a reduction involving loop invariant terms, back-substitution can be streamlined. Consider the case where each of the terms a_i in the previous example are identical values, say a. Assume also that a multiplicative operation $\times$ exists such that:

$$\sum_1^n a = n \times a$$

As an example, the associative operation + could represent floating point multiplication and the multiplicative operation × could be the n^{th} power of a. We start with the original code as modified to reflect the invariance of the a_i terms:

```
s[1] = s_in
do i=1,n
s = s[1] + a
remap(s)
enddo
```

Original code with loop invariant terms

Again, we assume that the initial value s_{in} resides in s[1] and, the final value s_{out} may be obtained in s[1] after loop execution. We now rewrite the code exactly as in the symmetric back-substitution case but pre-compute the addition of b copies of the invariant term a outside the loop. Again, this method is suitable for either reduction to scalar or reduction to vector, but now the computational work is a constant function of the degree of height reduction. This provides an excellent method for stepping out symmetric induction variables within software pipelined loops.

```
c = a × b
s[1] = s_in

do i=1,b-1
s = s[1] + a
remap(s)
enddo

do i=b,n
s = s[b] + c
remap(s)
enddo
```

Symmetric back-substitution with invariant terms

The invariant case eliminates the need for redundancy in the back-substituted expression resulting in the following loop performance metrics: Ops / iter = 1 and RecMII = $\frac{\ell}{b}$. Note that for associative operations which have a well behaved inverse operation (e.g. additive inverse), the first loop can be eliminated and proper values for s[1], s[2], ...,s[b] can be pre-calculated before entering a back-substituted loop with trip count n.

2.3 Blocked Back-substitution

In this section, we present a new technique which unrolls loop bodies to achieve a more efficient solution. We call this technique blocked back-substitution because it uses an asymmetric treatment of code within a block of b consecutive iterations where b represents the block size or degree of unrolling. Blocked back-substitution frequently height reduces recurrences with a substantially lower increase in redundant operation count.

The schema is as follows. A loop body is unrolled b times. The first b-1 iterations of the body are simple copies of the original loop text. The final copy of the loop body is expressed using b-fold back-substitution directly in terms of values live in to the first of the b iterations. The expression for this back-substituted iteration is then height reduced. The height reduction is asymmetrically optimized to minimize the loop recurring value-in to loop recurring value-out path (s[b] to s[0]).

The b fold unrolling of loop text into a block complicates the treatment of arbitrary loop trip count. For counted loops (*e.g.*, DO loops in FORTRAN), we will condition loop execution so that the blocked loop executes an integral multiple of b source iterations. This is achieved by defining two trip count functions: e (the epilogue count) and k (the kernel count). Both are functions of the source trip count n and the degree of blocking b. Let:

$$e(n,b) = n \mod b$$

$$k(n,b) = n - e(n,b)$$

Blocked back-substitution techniques also work for WHILE-loops but only DO-loops will be discussed here.

Using the conditioning approach described above, we parallelize the reduction recurrence applying blocking as shown below. Consider the first loop in the code. In this blocked loop, we call each of the unrolled iterations a minor iteration corresponding to a single iteration of the source program. Each body of the unrolled text consisting of b minor iterations is called a major iteration. Every b^{th} minor iteration (the last minor iteration in the unrolled loop body) is calculated using b fold back-substitution. Intervening minor iterations are calculated using an unmodified, non-height reduced recurrence. This eliminates redundant work within b-1 of the b minor iterations.

```
s[1] = s_in
do i=b,k(n,b),b
 s = s[1] + a_i          /* first minor iteration */
remap(s)
 ⋮
 s = s[1] + a_{i+b-2}    /* (b-1)th minor iteration */
remap(s)
 s = s[b] + (a_{i+b-1} + a_{i+b-2} + a_{i+b-3} + ··· + a_i)   /* bth minor iteration */
remap(s)
enddo

do i=e(n,b),n
 s = s[1] + a_i          /* retire residual iterations conventionally */
remap(s)
enddo
```

Blocked back-substitution

It may appear superficially that this code is not very parallel in that each of the first b-1 minor iterations depends upon a result calculated within the previous minor iteration. This creates a long chain of operations linked through flow dependence. As a result of the b fold back-substitution, the b^{th} minor iteration within the unrolled loop body depends only on values produced by that same code in the previous major iteration. Since the b^{th} minor iteration does not depend on any of the previous b-1 minor iterations, the only recurrence cycle in the loop depends on the relationship between this b^{th} minor iteration and the same code from the previous major iteration.

Another barrier to parallelization might appear to be the remap operations positioned between each of the minor iterations; however, that is not the case. A remap operation between two minor iterations doesn't act as a barrier for code motion. An operation can be moved across a remap operation (in either direction) with a corresponding adjustment to its EVR indices. This has no detrimental effect on program semantics if all dependencies are honored. Also, multiple remap operations within a loop body don't impose any constraints on the ResMII or the RecMII of the loop. For example, it may appear that a loop body with n remaps takes a minimum of n cycles (one per remap). On processors with no hardware support for the EVR concept, this is not an issue since remaps do not appear in the generated code. The paper by Rau *et.al.* [17] describes how to generate code for such processors by using kernel-unrolling and variable-renaming. For processors that do support the EVR concept in hardware, we have two options. The first option is to generalize the concept of the remap operation so that a single operation can perform remap by multiple positions. The second is to use a single remap by 1 at the end of each major iteration and use variable-renaming to remove other remap operations.

The technique is applicable in both reduction to scalar and reduction to vector type of recurrences. Note that in the reduction to scalar case, dead code elimination should be used to eliminate unused operations in the b-1 non-height reduced minor iterations of the loop. Loop performance metrics are calculated for the reduction to vector case as follows: Ops / iter = $\frac{2b-1}{b}$ and RecMII = $\frac{\ell}{b}$. In both symmetric and blocked back-substitution, additional parallelism can be exposed by increasing the degree of back-substitution b. However, while symmetric back-

substitution requires work which increases linearly with the degree of back-substitution b, blocked back-substitution requires work that is bounded by a two fold increase irrespective of b.

When terms are loop invariant, blocked back-substitution may take advantage of additional savings in redundant computation.

```
c = a × b
s[1] = s_in
do i=1,n,b
s = s[1] + a
remap(s)
s = s[1] + a
remap(s)
⋮
s = s[1] + a  /*unroll b-1 copies of original body*/
remap(s)
s = s[b] + c /* generate specialized final iteration*/
enddo
```

Blocked back-substitution with invariant terms

This approach is able to provide increasing degree of parallelism with no increasing work as indicated by the loop performance metrics: Ops / iter = 1 and RecMII = $\frac{\ell}{b}$. This technique can be used for the parallelization of the calculation of induction variables.

3 First Order Multiply-add Recurrences

We use the first order multiply-add recurrence to help illustrate the relative benefits of the height reduction approaches described above. The first order multiply add recurrence schema treat recurrences of the requisite form where primitive operations have the following properties: associative property of addition and the distributive property of multiplication over addition. While this technique is frequently applied to first order linear recurrences, the technique also applies to Boolean non-linear recurrences or any other recurrence with requisite form and properties.

The code presented below illustrates the form of the first order multiply-add recurrence.

```
s[1] = s_in
do i=1,n
s = a_i s[1] + c_i
enddo
```

Original first order multiply-add recurrence

In its unmodified form, the first order multiply-add recurrence has the following loop performance metrics: Ops / Iter = 2 and RecMII = 2ℓ. Here, we have assumed that the multiply and addition units have identical latency ℓ.

3.1 Symmetric Back-substitution

The expressions shown below illustrate the derivation of the back-substituted form for the first order linear recurrence. The unmodified expression shown on the first line evaluates s directly in terms of s[1]. The second line is derived by repeatedly (for k=1,..,b) substituting the expression for s[k] into the original expression and raising the order of the recurrence. In the final line representing the b^{th} order recurrence executed within the back-substituted loop, we have used the distributive property to isolate s[b]. This asymmetric height reduction calculates a new s from a b^{th} previous s after only a single multiplication and addition.

$$s = a_i s[1] + c_i$$

$$s = a_i(a_{i-1}(a_{i-2}(\cdots(a_{i-b-2}(a_{i-b-1}s[b] + c_{i-b-1}) + c_{i-b-2})\cdots) + c_{i-2}) + c_{i-1}) + c_i$$

$$s = (a_i a_{i-1}\cdots a_{i-b+1})s[b] + a_i(a_{i-1}(a_{i-2}(\cdots(a_{i-b+2}(c_{i-b+1}) + c_{i-b+2})\cdots) + c_{i-2}) + c_{i-1}) + c_i$$

The schema given below shows the code after symmetric back-substitution. As in section 2.2, the first loop initializes the recurrence. After initialization, the second loop repeatedly executes the back-substituted expression in order to accelerate the solution of the original recurrence.

```
s[1] = s_in
do i=1,b-1      /* treat b-1 iterations conventionally */
 s = a_i s[1] + c_i
remap(s)
enddo

do i=b,n        /* b fold back-substitution of remaining iterations */
 s = (a_i a_{i-1} ··· a_{i-b+1}) s[b]
     + a_i(a_{i-1}(a_{i-2}(···(a_{i-b+2}(c_{i-b+1}) + c_{i-b+2})···) + c_{i-2}) + c_{i-1}) + c_i
remap(s)
enddo
```

Symmetric back-substitution for first order multiply-add recurrence

The loop performance metrics in this case are as follows: Ops / iter = $(3b-1)$ and RecMII = $\frac{2\ell}{b}$. Thus, the number of operations required to execute an iteration grows linearly as the back-substitution distance is increased. This disadvantage restricts the method to a limited degree of back-substitution.

3.2 Blocked Back-substitution

The following schema shows the code for the first order multiply-add recurrence after blocked back-substitution. As in section 2.3, the loop is conditioned using the kernel and epilog trip count functions. The loop performance metrics in this case are as follows: Ops / iter = $\frac{5b-3}{b}$ and RecMII = $\frac{2\ell}{b}$

```
s[1] = s_in
do i=b,k(n,b),b
 s = a_i s[1] + c_i
remap(s)
 ⋮
 s = a_{i+b-2} s[1] + c_{i+b-2}
remap(s)
 s = (a_{i+b-1} a_{i+b-2} ··· a_i) s[b]
     + (a_{i+b-1}(a_{i+b-2}(a_{i+b-3}(···(a_{i-1}(c_i) + c_{i+1})···) + c_{i+b-3}) + c_{i+b-2}) + c_{i+b-1})
remap(s)
enddo

do i=e(n,b),n
 s = a_i s[1] + c_i
remap(s)
enddo
```

Blocked back-substituted first order multiply-add recurrence

4 Higher Order Recurrences

In this section, we consider higher order recurrences and show how the symmetric and blocked back-substitution techniques discussed earlier can be applied to reduce the height of these recurrences. As in the last section, we phrase the description in terms of two generic operations, called add and multiply, and rely only on the following properties of these operations: associativity of addition and multiplication and the distributivity of multiplication over addition. Thus, the results apply not only to higher-order linear recurrences, but to any higher order recurrence involving two operators with the required properties, *e.g.*, Boolean (non-linear) recurrences.

The code given below shows the general form of the type of recurrences that we will consider in this section. Furthermore, we will focus on the reduction to vector case, *i.e.*, the case where all n values of s are needed.

```
s[1] = s_1; ··· s[m] = s_m
do i = 1, n
 s = a_{1,i} s[1] + ··· + a_{m,i} s[m] + c_i
remap(s)
enddo
```

Original m^{th} order recurrence

In the first few subsections, we describe a framework for handling recurrences of the form shown above. We describe schemas for symmetric and blocked back-substitution, and show how performance metrics, *i.e.*, ops/iter and RecMII, can be computed for various schemas. These subsections don't provide any specific numbers or formulas for performance metrics. The reason

is that actual numbers or formulas depend upon what one assumes about the coefficients. However, the methodology presented in these subsections can be used to calculate performance metrics for various cases.

The remaining subsections provide formulas for performance metrics for several interesting cases listed below.

1. Recurrences in which all coefficients have non-zero loop-variant values.
2. Recurrences in which all coefficients have non-zero but loop-invariant values.
3. Recurrences in which $c_i = 0$ and the remaining coefficients have non-zero values. We consider both loop-variant and loop-invariant cases for coefficients.

4.1 Formulas for Coefficients in Back-substituted Expressions

This subsection derives formulas for coefficients in the expression that one gets after back-substitution is repeated to a certain degree. These formulas are applicable to both symmetric and blocked back-substitution techniques.

In the original program, the computation of s in the i^{th} iteration depends upon s[1] computed in the previous iteration. The back-substitution process involves substituting the expression for s[1] in the expression that computes s, and re-arranging the terms so as to minimize the heights of recurrence paths

$$s = a_{1,i}\left(a_{1,(i-1)}s[2] + \cdots + a_{m,(i-1)}s[m+1] + c_{i-1}\right) + a_{2,i}s[2] + \cdots + a_{m,i}s[m] + c_i$$

$$s = \left(a_{1,i}a_{1,(i-1)} + a_{2i}\right)s[2] + \cdots + \left(a_{1,i}a_{(m-1),(i-1)} + a_{m,i}\right)s[m] + a_{1,i}a_{m,(i-1)}s[m+1] + (a_{1,i}c_{i-1} + c_i)$$

The structure of the new expression is similar to the original expression except for the structure of coefficients and the values of s used in the expression. We can repeat the process by substituting for s[2], then for s[3] and so on. In the following discussion, we denote the coefficients in the expression for s obtained after a certain number of back-substitution by $A^b_{1,i} \cdots A^b_{m,i}$ and C^b_i; the superscript denotes the degree of back-substitution. The base case, *i.e.*, the original expression with no substitution performed is denoted by b=1. Using this notation, the expression for s after (b-1) fold back-substitution can be written as follows:

$$s = A^{b-1}_{1,i}s[b-1] + \cdots + A^{b-1}_{m,i}s[b+m-2] + C^{b-1}_i$$

To derive the expression for s after b fold back-substitution, we substitute the value of s[b-1] in the above expression. In the original program, s[b-1] is computed using the following expression:

$$s[b-1] = a_{1,(i-b+1)}s[b] + \cdots + a_{m,(i-b+1)}s[b+m-1] + c_{i-b+1}$$

Substituting the expression for s[b-1] and rearranging the terms gives the following expression for s.

$$s = \left(a_{1,(i-b+1)}A_{1,i}^{b-1} + A_{2,i}^{b-1}\right)s[b] + \cdots + \left(a_{(m-1),(i-b+1)}A_{1,i}^{b-1} + A_{m,i}^{b-1}\right)s[b+m-2]$$
$$+ \left(a_{m,(i-b+1)}A_{1,i}^{b-1}\right)s[b+m-1] + \left(c_{i-b+1}A_{1,i}^{b-1} + C_i^{b-1}\right)$$

Using the notation introduced earlier, the expression for s after b fold back-substitution can also be written as:

$$s = A_{1,i}^{b}s[b] + \cdots + A_{m,i}^{b}s[b+m-1] + C_i^{b}$$

Comparing the above two expressions for s, we get the following recursive formulas for coefficients. In these formulas, $A_{1,i}^{1} \cdots A_{m,i}^{1}$ and C_i^1 represent the base case (*i.e.*, b =1) and their values are simply the coefficients in the original expression for s.

$$A_{1,i}^{1} = a_{1,i} \qquad A_{1,i}^{b} = a_{1,(i-b+1)}A_{1,i}^{b-1} + A_{2,i}^{b-1}$$
$$\vdots \qquad\qquad \vdots$$
$$A_{(m-1),i}^{1} = a_{(m-1),i} \qquad A_{(m-1),i}^{b} = a_{m,(i-b+1)}A_{1,i}^{b-1} + A_{m,i}^{b-1}$$
$$A_{m,i}^{1} = a_{m,i} \qquad A_{m,i}^{b} = a_{m,(i-b+1)}A_{1,i}^{b-1}$$
$$C_i^1 = c_i \qquad C_i^b = c_{i-b+1}A_{1,i}^{b-1} + C_i^{b-1}$$

It is important to note that these formulas don't have the same structure. Specifically, the second to last formula involves only multiplications and no additions. For a specific value of m, the formulas are used in the reverse order. For example, the first order case uses the last two formulas, the second order last three and so on.

In both symmetric and blocked back-substitution, the work involved in computing coefficients is a significant amount of the overall work. To minimize the work, it is important to recognize common sub-expressions. For example, the computation of $A_{1,i}^{b-1}$ is common to all the formulas given above. It is also important to factor common sub-expressions, *i.e.*, compute ab + ac as a(b+c). We believe the formulas given above compute coefficients using a minimum number of add and multiply operations, but we offer no formal proof of this fact.

As an example of the application of these formulas, consider the first order case. For the symmetric back-substitution with b = 3, the computation of s can be expressed using the following set of equations.

$$A_{1,i}^{3} = a_{1,(i-2)}A_{1,i}^{2}; \quad A_{1,i}^{2} = a_{1(i-1)}a_{1,i}$$
$$C_i^3 = c_{i-2}A_{1,i}^{2} + a_{1i}c_{i-1} + c_i$$
$$s = A_{1,i}^{3}s[3] + C_i^3$$

These equations compute the same result as the one given in Section 3.1 and use the same number of operations (which is optimal). They are simply an alternate way of organizing the computation than the one given in Section 3.1.

4.2 Counting Operations

The general method to calculate the number of operations in the expression obtained after b fold back-substitution is to express number of operations as a recurrence of the following form.

$$Ops_1 = p; \quad Ops_b = Ops_{b-1} + q$$

In these equations, p is the number of operations in the original expression, b is the degree of back-substitution, and q is the number of additional operations needed to compute coefficients for b fold back-substitution given the coefficients for (b-1) fold back-substitution. In other words, q is simply the number of explicitly displayed operations in the formulas for coefficients given in Section 4.1. Recurrence relations of this form can be used not only to compute total number operations, but also to compute total number of operations of a certain type, *e.g.*, add operations.

4.3 Symmetric Back-substitution

The schema for the code generated by the symmetric back-substitution technique is shown below. As in the previous sections, the first sequential loop initializes the recurrence by computing the first b values of s, and the second loop uses the back-substituted expression to accelerate the recurrence.

$s[1] = s_1; \cdots s[m] = s_m$
do i = 1, b-1 /* Treat b-1 iterations conventionally */
$s = a_{1,i}s[1] + \cdots + a_{m,i}s[m] + c_i$
remap(s)
enddo

do i = b, n /* Back-substitute remaining iterations */
$s = A^b_{1,i}s[b] + \cdots + A^b_{m,i}s[b+m-1] + C^b_i$
remap(s)
enddo

Code schema for symmetric back-substitution

The RecMII of the main loop depends upon the order in which terms are added. Thus, it is important to find an order of summation that gives the minimum RecMII. Finding such an order, however, is a challenge and we don't provide a general solution that works in all cases. The general problem can be stated as follows. Suppose n_i is the number of operation on the path from s[b+i] to s. Then,

$$RecMII = Max\left(\frac{\ell n_0}{b}, \frac{\ell n_1}{b+1}, \cdots, \frac{\ell n_{m-1}}{b+m-1}\right),$$

and we need to find the values of $n_0, n_1, \cdots$ that minimizes the RecMII. Note that the optimal solution not only depends on b and m but also on the nature of coefficients (0, 1, loop-invariant etc.) in the original expression.

The two approaches that give near optimal results are as follows. The first is to sum the terms from right to left. This gives near optimal results for small values of b, which is the important case for symmetric back-substitution. The other approach is to sum the terms using a log-tree. This works well for higher values of b, since it becomes important for higher values of b to jointly minimize the lengths of all recurrence paths.

4.4 Blocked Back-substitution

The schema for the code generated by the blocked back-substitution technique is given below; see Section 2.3 for the definitions of k(n,b) and e(n, b). The first loop handles iterations that are multiple of b; the second loop executes the remaining iterations sequentially.

$s[1] = s_1; \cdots s[m] = s_m$
do i = 1, k(n, b), b
$s = a_{1,i}s[1] + \cdots + a_{m,i}s[m] + c_i$
remap(s)
⋮
$s = a_{1,(i+b-m-1)}s[1] + \cdots + a_{m,(i+b-m-1)}s[m] + c_{(i+b-m-1)}$
remap(s)
$s = A^{b-m+1}_{1,(i+b-m)}s[b-m+1] + \cdots + A^{b-m+1}_{m,(i+b-m)}s[b] + C^{b-m+1}_{i+b-m}$
remap(s)
⋮
$s = A^{b}_{1,(i+b-1)}s[b] + \cdots + A^{b}_{m,(i+b-1)}s[b+m-1] + C^{b}_{i+b-1}$
remap(s)
enddo

do i = e(n, b), n
$s = a_{1,i}s[1] + \cdots + a_{m,i}s[m] + c_i$
remap(s)
enddo

Code schema for blocked back-substitution

The first loop is derived from the original loop as follows. The original loop is unrolled b times. As in the last section, we call each unrolled iteration a minor iteration and call each iteration of the unrolled text consisting of b minor iteration a major iteration. The last m minor iterations are expedited using the back-substitution. The last minor iteration is calculated using b fold back-substitution, the second last using (b-1) fold back-substitution etc. Each of the last m minor iterations in a major iteration is computed as a function of last m values of s computed by the previous major iteration. Another way to view the loop is that each major iteration takes m values from the previous iteration and computes m values in a fast way for the next iteration. The remaining b-m minor iterations in a major iteration are calculated using the original recurrence, eliminating redundant work within these iterations.

An important point to note is that it is not sufficient to expedite only the last minor iteration using b fold back-substitution. The reason is that, after back-substitution, the last minor iteration

uses m values of s computed in the previous major iteration. Expediting only the last minor iteration forces m-1 of these values to be computed sequentially in the previous major iteration, which forces the entire loop to run more or less sequentially.

In the code schema given above, we assume that $b \geq m$, that is the loop is unrolled at least as many times as the order of the recurrence. Applying the blocked back-substitution technique with $b < m$ seems similar to applying the symmetric back-substitution technique and then unrolling the loop. Further work is needed to see if the blocked back-substitution technique offer any advantages over the symmetric back-substitution when $b < m$.

As in the symmetric back-substitution technique, the RecMII of the main loop depends upon the order in which terms are added in each of the expedited iterations. On the other hand, the RecMII doesn't depend upon the summation order used in the first b-m minor iterations, since computation in any of these iterations is not on a critical path. Thus, we will use the right to left summation order for the first b-m iterations.

For expedited iterations, it is sufficient to find a summation order that minimizes the RecMII of one of the expedited iteration, since all expedited iterations calculate expressions with identical top-level structure. Consider one of the last b-m minor iteration. Suppose n_i is the number of operation on the path from s[b+i] to s. Since all values of s used in the computation are produced in the previous major iteration, the RecMII for the minor iteration is given by

$$\text{RecMII} = \text{Max}(\ell n_0, \ell n_1, \cdots, \ell n_{m-1}).$$

And we need to find the values of $n_0, n_1, \cdots$ that minimizes the RecMII. The optimal solution depends on m and on the nature of coefficients (0, 1, loop-invariant etc.) in the original expression, but it doesn't depend upon b. The general strategy to minimize RecMII is to find a summation order in which $n_0 \cdots n_{m-1}$ are all equal. Thus, the log-tree approach for summing the terms gives near optimal results in most cases.

4.5 Recurrences with Non-zero Loop-variant Coefficients

In this subsection, we present performance metrics for the case when all the coefficients have non-zero loop-variant values. We consider two different architectural models, which are described below.

1. The first architecture provides pipelined functional units that can perform add and multiply operations. We assume that an operation can be issued on an unit at each cycle and that the operation latency is ℓ cycles.
2. The second architecture provides pipelined functional units that can perform not only add and multiply operations but also fused multiply-add operation. Again, we assume that an operation can be issued on an unit at each cycle and that the operation latency is ℓ cycles. The discussion in the context of this architecture is more appropriate for arithmetic or linear recurrences as some of the recent architectures provide fused multiply-add operation for floating-point numbers.

Table 1 summarizes the performance metrics for the original program and for the programs obtained after applying the symmetric and the blocked back-substitution techniques. The formulas in the table are in terms of the following variables: the order of the original recurrence

m, the degree of back-substitution b, and the latency ℓ. For blocked back-substitution, b is also the number of times the loop is unrolled and is assumed be greater than or equal to m. The rest of this subsection gives more details about the performance metrics.

Table 1: Performance metrics for the case of non-zero loop-variant coefficients. In the formulas, m is the order of the original recurrence, b is the degree of back-substitution and ℓ is the latency.

Technique	With support for + and × only		With support for +, × and fused multiply-add	
	RecMII	Ops/iter	RecMII	Ops/iter
Original	2ℓ	$2m$	ℓ	m
Symmetric	$\frac{\ell(m+1)}{b+m-1}$	$b(2m+1)-1$	$\frac{\ell m}{b+m-1}$	$b(m+1)-1$
Blocked $b \geq m$	$\frac{\ell}{b}\left(1+\lceil \log_2(m+1) \rceil\right)$	$\frac{1}{b}\begin{pmatrix} bm(2m+3) \\ -\frac{1}{2}m(m+1)(2m+1) \end{pmatrix}$	$\frac{\ell}{b}\left(1+\lceil \log_2 m \rceil\right)$	$m=1$: $\frac{1}{b}(3b-2)$ $m>1$: $\frac{1}{b}\begin{pmatrix} bm(m+2)+m\lfloor m/2 \rfloor \\ -\frac{1}{2}m\left(m^2+2m+3\right) \end{pmatrix}$

First, we discuss the performance metrics in the context of the architecture in which each unit is capable of performing multiply and add operations. For the original program, the right to left order of summing terms gives the minimum value of RecMII and is used to derive the value shown in Table 1. The number of operations performed in each iteration is simply 2m.

For symmetric back-substitution, we consider only the right to left order for summing terms. As pointed out earlier, this summation order gives near optimal results for small values of b, which are more relevant for this technique. For first and second order recurrences, there is no difference between the two approaches, i.e. the right to left order or the log-tree approach. For third order recurrences, the log-tree approach is useful for $b > 6$. For higher orders, b has to be even greater. Assuming the right to left order, RecMII is given by

$$\text{RecMII} = \text{Max}\left(\frac{2\ell}{b}, \frac{3\ell}{b+1}, \cdots, \frac{(m+1)\ell}{b+m-1}\right),$$

in which the last term is greater than or equal to all other terms for any $b > 1$. The number of operations performed in each iteration is determined using the methodology suggested in Section 4.2. The recurrence relation for this technique are

$$\text{Ops}_1 = 2m; \quad \text{Ops}_b = \text{Ops}_{b-1} + (2m+1),$$

which are derived by simply looking at the original expression and equations for coefficients given in Section 4.1. Solving this recurrence gives

$$\text{Ops}_b = (b-1)(2m+1) + 2m = b(2m+1) - 1.$$

For blocked back-substitution, the formula for RecMII assumes that terms in each of the expedited iterations are summed using the log-tree, which gives near optimal results in this case. In the formula, $\lceil \log_2(m+1) \rceil$ is the height of the log-tree with m+1 terms and 1 accounts for the multiplication performed on each path. The ops/iter metric is derived by summing the contribution of each of the minor iterations. The number of operations in each of the first b-m

iterations is identical to that in the original recurrence, which is 2m. For back-substituted iterations, the recurrence relation used in computing the number of operations turns out to be identical to the one given above for symmetric back-substitution; the summation order has no influence on the operation count. Thus, the number of operation in each of the back-substituted iterations can be obtained by instantiating the equation for Ops_b given above with an appropriate value of b. Therefore, we get

$$Ops / iter = \frac{1}{b}\Big((b-m)2m + (b(2m+1)-1) + \cdots + \big((b-(m-1))(2m+1)-1\big)\Big),$$

which gives us the formula shown in Table 1.

Now we discuss the performance metrics in the context of the architecture in which functional units are also capable of performing fused multiply-add. For such architectures, computation should be structured so that as many multiplications as possible are combined with additions without having an adverse impact on RecMII. Structuring computation of coefficients is easy because the computation is not on recurrence paths and because the formulas for most of the coefficients (except the second last) already have the multiply-add structure. The starting point for deriving the structure for the top-level expression is to consider the summation order that gives the minimum RecMII. In the following discussion, we use MA to denote a multiply-add operation; its semantics is as follows.

$$MA(a,s,c) = a \times s + c$$

For both the original program and symmetric back-substitution, we use the right to left summation order for the top-level expression. The following general form shows how the multiplications are combined with additions in these cases; $a_1 \cdots a_m$ and c denote coefficients, and $s_1 \cdots s_m$ denote previous values of s.

$$MA(a_1, s_1, MA(a_2, s_2, \cdots MA(a_m, s_m, c) \cdots))$$

Thus, for the original program, the RecMII is ℓ, *i.e.*, the latency of a single multiply-add operation, and the number of operations per iteration is m.

For symmetric back-substitution, the RecMII is given by

$$RecMII = Max\left(\frac{\ell}{b}, \frac{2\ell}{b+1}, \cdots, \frac{m\ell}{b+m-1}\right)$$

in which the last term is greater than or equal to all other terms for any $b > 1$. The number of operations per iteration is computed using the following recurrence relation:

$$Ops_1 = m; \quad Ops_b = Ops_{b-1} + (m+1).$$

Solving this recurrence gives

$$Ops_b = (b-1)(m+1) + m = b(m+1) - 1.$$

Now we consider the blocked back-substitution. For the first b-m iterations, we assume that the summation order is right to left and multiplications are combined with additions in the way

shown above for the original recurrence. The starting point for back-substituted iterations is the log-tree approach for summing terms. Combining multiplies with adds in this case is a little more difficult. In the case of first order recurrences, the top-level of the expression simply requires one multiply-add. The general approach used for the second and higher orders is shown in Figure 1. The figure as shown applies to odd values of m. Note that incorporating the leftmost pair of values in the log-tree may introduce a new level in the tree. This happens when m is of the form $2^n + 1$ for some n. For even values of m, the leftmost term is omitted and the log-tree contains only add operations.

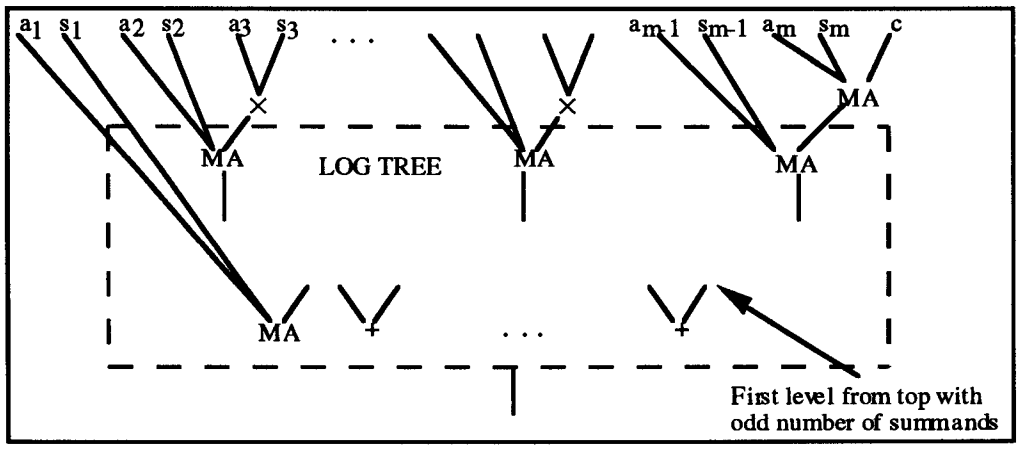


Figure 1: The structure of the top-level expression assumed in computing performance metrics for blocked back-substitution in the context of architectures that provide fused multiply-add capability.

To derive the performance metrics, we need to compute the height of the log-tree and the number of operations within the log-tree in Figure 1. The easiest way to do this is to assume that each multiply-add operation inside the log-tree is just an add operation and to assume that the first two operands of each multiply-add inside the log-tree constitute a single operand. For example, the leftmost multiply-add in the top level of the log-tree is assumed to be an add operation and the first two operands, a_2 and s_2 are assumed to be a single operand to the add operation. Then, the log-tree has m inputs and combines them using the dyadic add operation. Therefore, the height of the log-tree is $\lceil \log_2 m \rceil$, and the number of operation within the tree is m-1.

Therefore, the height of the entire expression, and hence the RecMII of the loop is $1+\lceil \log_2 m \rceil$; 1 accounts for the level above the log-tree.

The ops/iter metric is derived by summing the contribution of each minor iteration. First, we consider the first order case. In this case, the operation count for each of the first b-1 minor iterations is simply 1, and the operation count for the back-substituted iteration is given by the following recurrence:

$$Ops_1 = 1; \quad Ops_b = Ops_{b-1} + 2.$$

Solving this recurrence yields $Ops_b = 2b - 1$. By summing the contribution of each minor iteration, we get

$$Ops / iter = \tfrac{1}{b}\left((b-1)1 + 2b - 1\right) = \tfrac{1}{b}(3b-2).$$

Now we consider second and higher order cases. The operation count for each of the first b-m minor iterations is m. The operation count for a back-substituted iteration can be derived using the following recurrence:

$$Ops_1 = \lfloor \tfrac{m}{2} \rfloor + m - 1; \quad Ops_b = Ops_{b-1} + (m+1).$$

In the equation for Ops_1, the first term is the number of operations in the level above the log-tree in Figure 1 and m-1 is the number of operations in the log-tree itself. The solution of this recurrence is

$$Ops_b = b(m+1) - 2 + \lfloor \tfrac{m}{2} \rfloor.$$

The operation count for each of the back-substituted iterations can be derived by instantiating the above equation with an appropriate value of b. By summing the contribution of each minor iteration, we get

$$Ops / iter = \tfrac{1}{b}\left((b-m)m + \left(b(m+1) - 2 + \lfloor \tfrac{m}{2} \rfloor\right) + \cdots + \left((b-(m-1))(m+1) - 2 + \lfloor \tfrac{m}{2} \rfloor\right)\right),$$

which upon simplification yield the formula shown in Table 1.

4.6 Recurrences with Non-zero Loop-invariant Coefficients

In this subsection, we present performance metrics for one of the important special case, namely, the case when all the coefficients are loop-invariant values. In this case, the computation of coefficients can be moved out of the loop. Thus, both symmetric and blocked back-substitution pay no penalty in increased operation count. As a result, both techniques offer substantial performance improvements in this case even for machines with low degree of parallelism.

Table 2 summarizes the performance metrics for this case. The formulas for RecMII remain the same as in the last section(see Table 1), since moving the computation of coefficients out of the loop has no impact on the RecMII of the loop. As expected, the number of operations per iterations is substantially lower than in the last section. For symmetric back-substitution, it is identical to that for the original program. For blocked back-substitution, it is identical to the original program for architectures with add and multiply capabilities but slightly higher than the original program for architectures with fused multiply-add capability. The reason for higher operation count is that log-tree summation order used in the case of blocked back-substitution is not as suited as right to left order for combining multiplications with additions.

Table 2: Performance metrics for the case of non-zero loop-invariant coefficients. In the formulas, m is the order of the original recurrence, b is the degree of back-substitution and ℓ is the latency.

Technique	With support for + and × only		With support for +, × and fused multiply-add	
	RecMII	Ops/iter	RecMII	Ops/iter
Original	2ℓ	2m	ℓ	m
Symmetric	$\frac{\ell(m+1)}{b+m-1}$	2m	$\frac{\ell m}{b+m-1}$	m
Blocked $b \geq m$	$\frac{\ell}{b}\left(1+\lceil \log_2(m+1) \rceil\right)$	2m	$\frac{\ell}{b}\left(1+\lceil \log_2 m \rceil\right)$	m=1: 1 m>1: $\frac{1}{b}\left(bm + m\lfloor m/2 \rfloor - m\right)$

4.7 Recurrences with $c_i = 0$

In this section, we discuss the case when the coefficient c_i is zero and the remaining coefficients have non-zero values. We consider both loop-invariant and loop-variant cases for the remaining coefficients and present performance metrics in the context of the two architecture models described in Section 4.5.

Note that first order recurrences in this case have a somewhat different structure than second and higher order recurrences. First order recurrences have the form of a product whereas second and higher order recurrences have the form of a sum of products.

Table 3 summarizes the performance metrics for the case when all the coefficients except c_i are loop-variant values. The derivation of these formulas parallels the derivation of the formulas presented in Section 4.5, and the assumptions made in that section also apply to this section. Thus, we omit most of the details and discuss only the main points.

Table 3: Performance metrics for the case when $c_i = 0$ and the remaining coefficients have non-zero loop-variant values. In the formulas, m is the order of the original recurrence, b is the degree of back-substitution and ℓ is the latency.

Technique	With support for + and × only		With support for +, × and fused multiply-add	
	RecMII	Ops/iter	RecMII	Ops/iter
Original	$m=1$: ℓ $m>1$: 2ℓ	$2m-1$	ℓ	m
Symmetric	$m=1$: $\frac{\ell}{b}$ $m>1$: $\frac{\ell m}{b+m-2}$	$b(2m-1)$	$\frac{\ell m}{b+m-1}$	bm
Blocked $b \geq m$	$\frac{\ell}{b}\left(1+\lceil \log_2 m \rceil\right)$	$\frac{1}{b}\begin{pmatrix} b(m+1)(2m-1) \\ -\frac{1}{2}m(m+1)(2m-1) \end{pmatrix}$	$\frac{\ell}{b}\left(1+\lceil \log_2 m \rceil\right)$	$m=1$: $\frac{1}{b}(2b-1)$ $m>1$: $\frac{1}{b}\begin{pmatrix} bm(m+1)+m\lfloor m/2 \rfloor \\ -\frac{1}{2}m\left(m^2+m+2\right) \end{pmatrix}$

As in Section 4.5, we use the right to left summation order for the original program and for the symmetric back-substitution technique. A point to note about symmetric back-substitution is that there is no difference between the right to left order and the log-tree approach for first, second and third order recurrences. For fourth order recurrences, the log-tree approach is useful only for $b > 6$. For higher orders, b has to be even greater. Thus, the right to left order gives near-optimal result in all relevant cases. For blocked back-substitution, we use the log-tree approach for the back-substituted minor iterations and the right to left order for the remaining iterations.

First, we consider the performance metrics in the context of architectures in which functional units are capable of performing add and multiply operations. The only formula that needs some explanation is the one for RecMII in the case of symmetric back-substitution. Recall that we assume right to left summation order for symmetric back-substitution. Thus, for first order recurrences, there is only a multiplication on the recurrence path. For higher orders, the RecMII is given by

$$\text{RecMII} = \text{Max}\left(\frac{2\ell}{b}, \frac{3\ell}{b+1}, \cdots, \frac{m\ell}{b+m-2}, \frac{m\ell}{b+m-1}\right).$$

Since $c_i = 0$, the path from the last term in the summation to the result of the expression doesn't involve an additional add operation. Thus, the path from the last term to the result has the same length (*i.e.*, $m\ell$) as the path from the second last term to the result. In the equation for RecMII, the second to last term dominates all other terms for any $b > 1$.

Next, we consider the performance metrics in the context of architectures that provide fused multiply-add capability. We assume that both the original recurrence and the symmetric back-substitution technique use the following general form to combine multiplications with additions.

$$\text{MA}(a_1, s_1, \text{MA}(a_2, s_2, \cdots \text{MA}(a_{m-1}, s_{m-1}, a_m \times s_m) \cdots))$$

Then, the formulas for performance metrics follow easily. For example, the RecMII in the case of symmetric back-substitution is given by

$$\text{RecMII} = \text{Max}\left(\frac{\ell}{b}, \frac{2\ell}{b+1}, \cdots, \frac{(m-1)\ell}{b+m-2}, \frac{m\ell}{b+m-1}\right),$$

in which the last term dominates all other terms for any $b > 1$. In the case of blocked back-substitution, we assume the iterations method to combine multiplications with additions for the first b-m iterations is the same as the one shown above for the original recurrence. For the back-substituted iterations, multiplications are combined with additions in a way similar to that shown in Figure 1. The only difference is that the leftmost MA at the top level that computes $a_m s_m + c$ is replaced by a simple multiplication that computes $a_m s_m$.

Table 4 summarizes the performance metrics for the case when all the coefficients except c_i are non-zero loop-invariant values. As pointed out in Section 4.6, the computation of coefficients in this case can be moved out of the loop. Thus, both symmetric and blocked back-substitution techniques perform about the same number of operations per iteration as the original program. The formulas for RecMII are identical to those in Table 3.

Table 4: Performance metrics for the case when $c_i = 0$ and the remaining coefficients have non-zero loop-invariant values. In the formulas, m is the order of the original recurrence, b is the degree of back-substitution and ℓ is the latency.

Technique	With support for + and × only		With support for +, × and fused multiply-add	
	RecMII	Ops/iter	RecMII	Ops/iter
Original	$m=1$: ℓ $m>1$: 2ℓ	$2m-1$	ℓ	m
Symmetric	$m=1$: $\frac{\ell}{b}$ $m>1$: $\frac{\ell m}{b+m-2}$	$2m-1$	$\frac{\ell m}{b+m-1}$	m
Blocked $b \geq m$	$\frac{\ell}{b}\left(1+\lceil \log_2 m \rceil\right)$	$2m-1$	$\frac{\ell}{b}\left(1+\lceil \log_2 m \rceil\right)$	$m=1$: 1 $m>1$: $\frac{1}{b}\left(bm+m\lfloor m/2 \rfloor - m\right)$

5 Performance results for selected machines

To put in perspective the results and formulas given in previous sections, we present graphs illustrating the benefits of the symmetric and blocked back-substitution techniques for machines with varying number of functional units and varying operation latency. We consider first order recurrences with non-zero loop-invariant coefficients and consider architectures with functional units capable of performing add and multiply but not fused multiply-add.

Figure 2 presents the benefits of symmetric back-substitution over the original program for machines with 1 to 6 functional units assuming the operation latency ℓ to be 3 cycles. Part (a) of the figure shows the MII of the loop as a function of the degree of back-substitution b (note that b = 1 means the original program). Each curve corresponds to a different number of functional units, which is indicated at the end of the curve. Part (b) presents the same data in terms of speedup for a given machine, that is,

$$\text{speedup} = \frac{\text{MII of the original program for machine with u units and latency } \ell}{\text{MII of the program after back substitution for the same machine}}$$

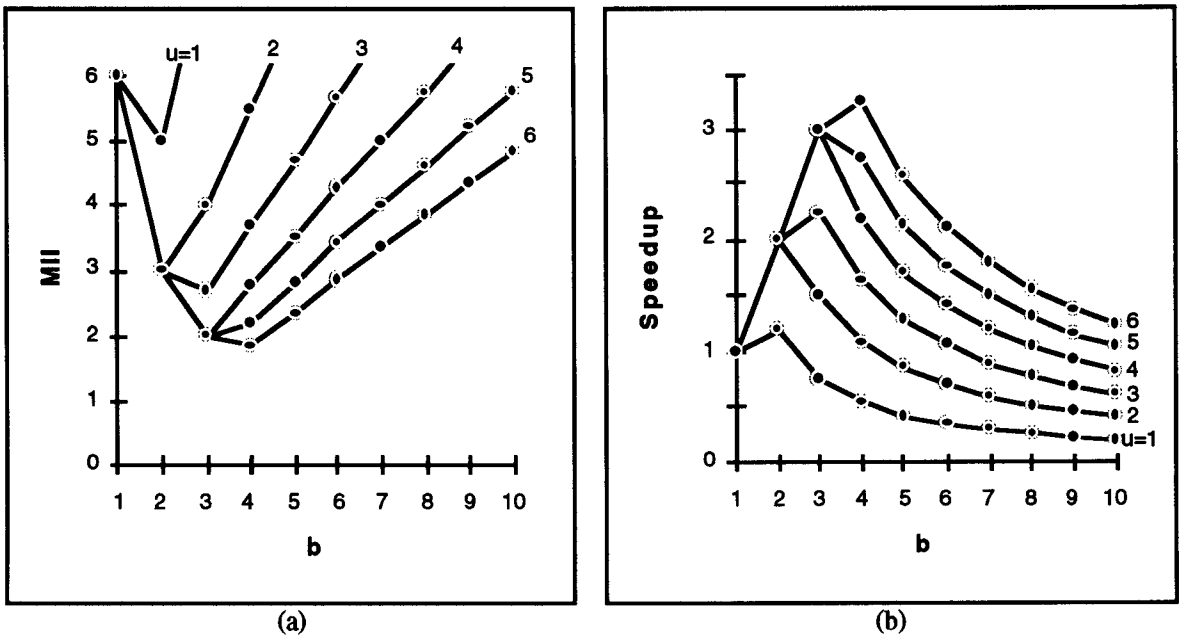
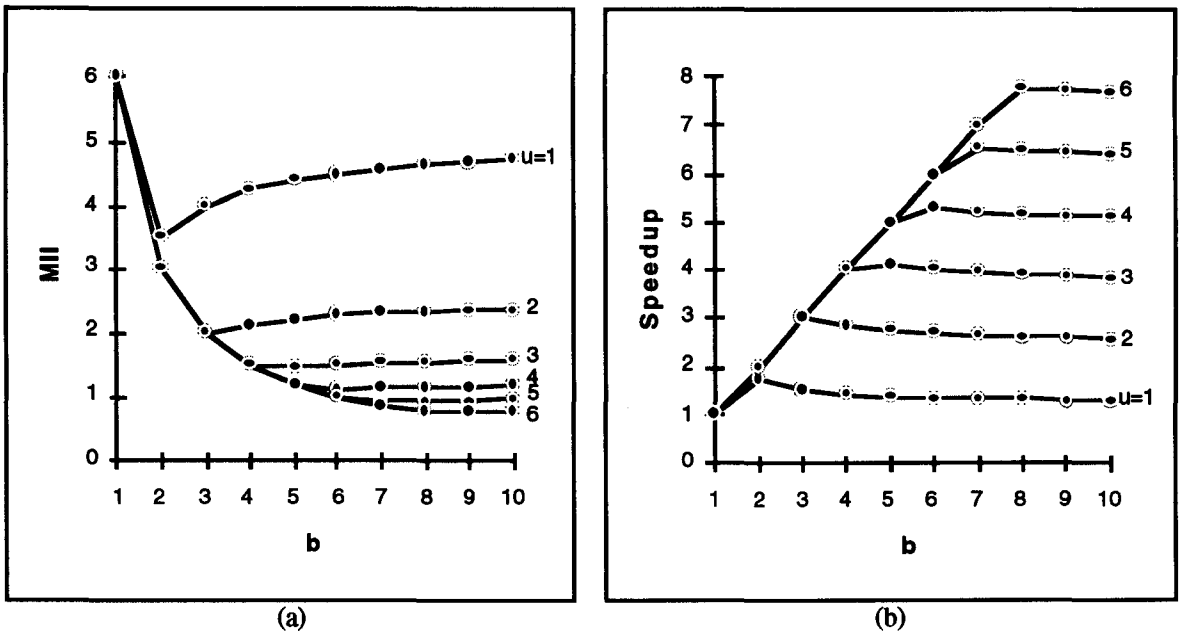


Figure 2: Graphs illustrating the benefits of symmetric back-substitution for machines with 1 to 6 functional units for the case of m = 1 and ℓ = 3. (a): MII of the loop after back-substitution as a function of b. (b): Speedup for a given machine as a function of b.

It is evident from Figure 2(a) that, for a given machine, there is an optimal degree of back-substitution b that gives the best performance benefits. This b corresponds to the case when both RecMII and ResMII of the loop after back-substitution are nearly equal. For b less than the optimal value, the RecMII of the loop dominates, and further back-substitution helps in reducing the RecMII. For b greater than the optimal value, the ResMII dominates, and further back-substitution actually decreases the performance.



(a) (b)

Figure 3: Graphs illustrating the benefits of blocked back-substitution for machines with 1 to 6 functional units for the case of m = 1 and ℓ = 3. (a): MII of the loop after back-substitution as a function of b. (b): Speedup for a given machine as a function of b.

Figure 3 illustrates the benefits of blocked back-substitution for machines with varying number of functional units under the same assumptions. Again, it is evident from Figure 3(a) that there is an optimal degree of back-substitution that gives the best performance for a given machine.

A couple of advantages of the blocked back-substitution technique are obvious from Figures 2 and 3. First, for a given machine, the blocked back-substitution technique provides better performance than the symmetric back-substitution. Second, the blocked back-substitution technique is more tolerant of over-substitution than the symmetric back-substitution. That is, for b greater than the optimal value, the performance degradation in the case of blocked back-substitution is not as large as in the case of symmetric back-substitution. Thus, exact determination of the optimal value of b is not as crucial in the case of blocked back-substitution as it is in the case of symmetric back-substitution.

Figure 4 illustrates the benefits of symmetric back-substitution by varying the operation latency but keeping the number of functional units fixed at 2. All other assumptions remain the same. The latency assumption for a curve is indicated by a number near the curve. As in the case of varying number of functional units, there is an optimal value of b for a given latency that gives the best performance. For values of b less than the optimal value, the loop is RecMII-limited; for values of b greater than the optimal value, the loop is ResMII-limited. All the curves in Figure 4(a) reach the same value of MII. This simply reflects the fact that the loop is ResMII-limited and the ResMII of a loop doesn't depend upon the operation latency. As pointed out in previous sections, symmetric back-substitution has the disadvantage that ops/iter increases linearly with the degree of back-substitution b. Figure 4(a) shows this fact clearly; the MII of the loop in the ResMII-limited region increases linearly with b.

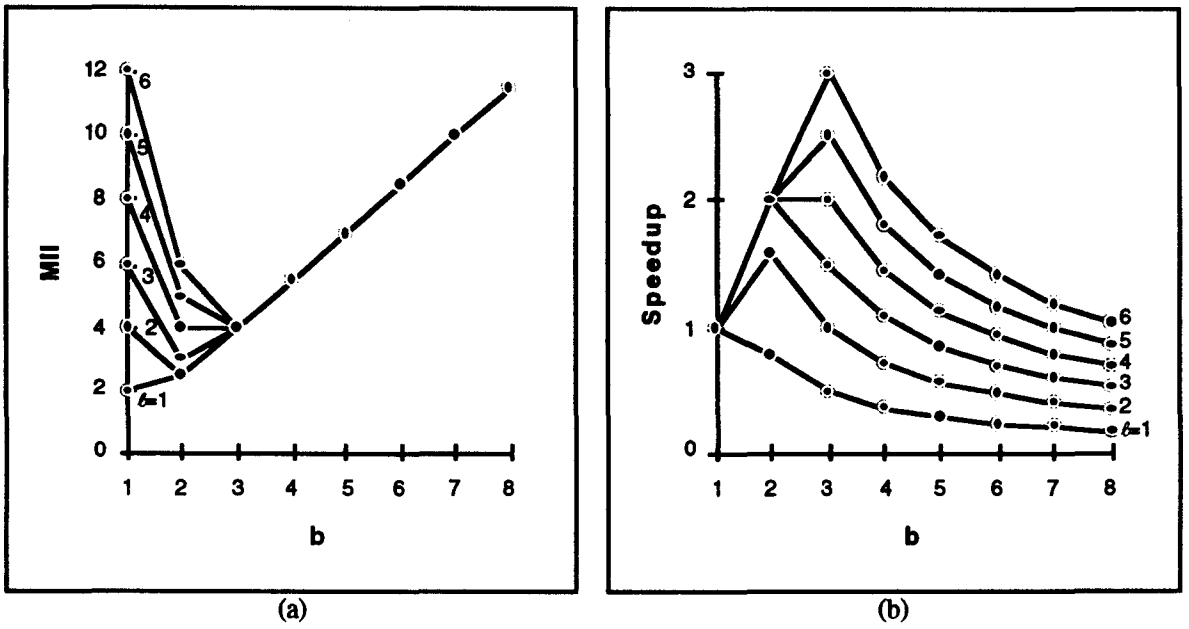


Figure 4: Graphs illustrating the benefits of symmetric back-substitution with operation latency varying from 1 to 6 for the case of m = 1 and number of units = 2. (a): MII of the loop after back-substitution as a function of b. (b): Speedup for a given machine as a function of b.

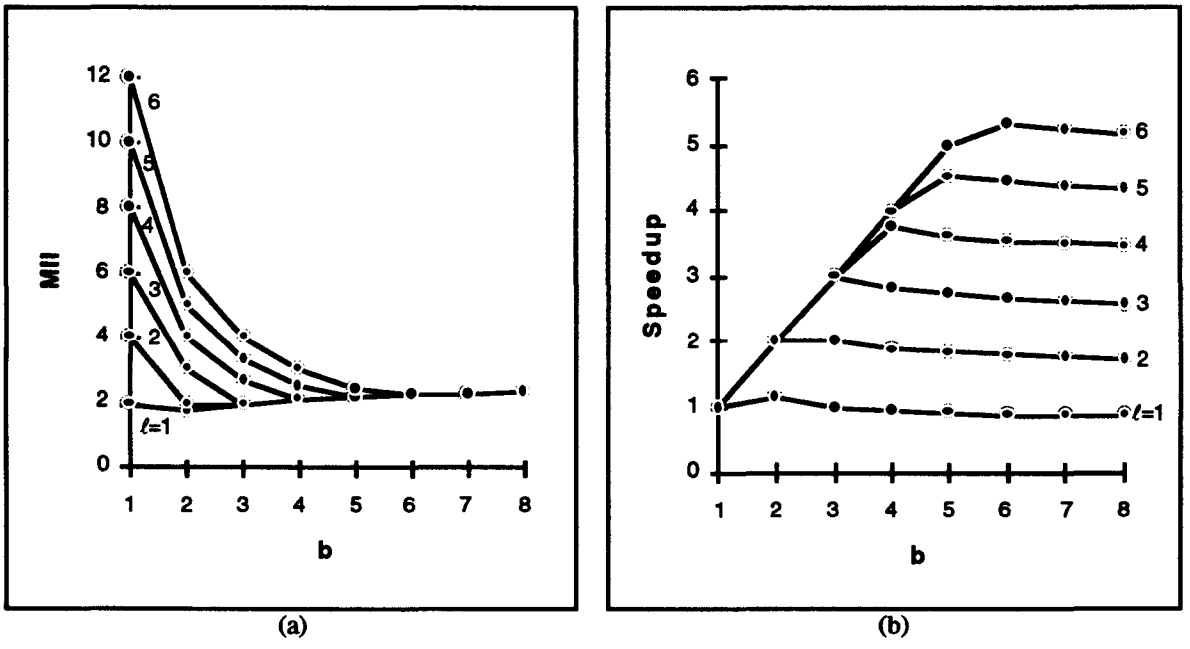


Figure 5: Graphs illustrating the benefits of blocked back-substitution with operation latency varying from 1 to 6 for the case of m = 1 and number of units = 2. (a): MII of the loop after back-substitution as a function of b. (b): Speedup for a given machine as a function of b.

Figure 5 illustrates the benefits of blocked back-substitution by varying the operation latency but keeping the number of functional units fixed at 2. Most of the comments made in the context of

symmetric back-substitution also apply here. Blocked back-substitution has the advantage that ops/iter remains nearly constant and doesn't increase linearly with the degree of back-substitution b. Figure 5(a) shows this fact clearly; the MII of the loop in the ResMII-limited region is nearly constant.

Figure 6 is an attempt to quantify the areas of applicability of the two techniques. It plots the best speedup that can be obtained as a function of the number of functional units for first, second and third order recurrences. We assume the following: recurrences have non-zero loop-variant coefficients, architecture doesn't provide fused multiply-add capability, and operation latency is 3. Each point in the plot is annotated with the technique and the optimal value of b that achieves that speedup. In an annotation, the letter denotes the technique that gives the best speedup–O stands for the original program, S for the symmetric back-substitution, and B for the blocked back-substitution. The number in an annotation denotes the optimal value of b.

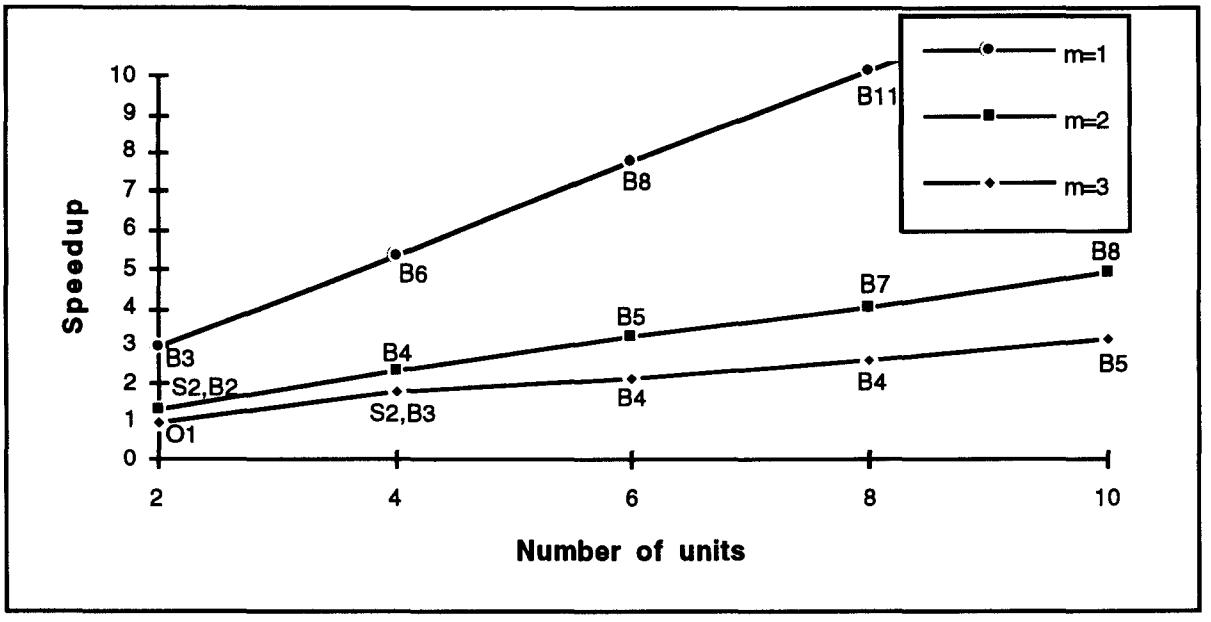


Figure 6: The best speedup that can be achieved for first, second, and third order recurrences. Each point is annotated with the technique and the optimal value of b. Operation latency is 3.

Figure 6 shows that the choice of the appropriate technique depends upon both the order of the recurrence and the parallelism provided by the machine. For first and second order recurrences, blocked back-substitution is the best technique under the assumptions mentioned above. For the third order case, the speedup curve shows that there are three distinct regions based on the parallelism in the machine; the regions are not explicitly shown in the figure but can be deduced by looking at the labels. In the first region, neither of the back-substitution techniques have much to offer. In the second region, symmetric and blocked back-substitution provide similar benefits. In the third region, blocked back-substitution is the best technique. Although the figure doesn't show it, there is also a fourth region where symmetric back-substitution performs better than blocked back-substitution. This happens when the order of the recurrence is high and the machine provide only a moderate amount of parallelism. In this case, the optimal value of b is very low (2 or 3). However, blocked back-substitution technique as defined in this paper requires b to be greater than m. Symmetric back-substitution, on the other hand, has no such limitation.

The actual speedup numbers presented in Figure 6 are, in some sense, a lower bound on speedup because they assume the most general form for coefficients, *i.e.*, non-zero loop-invariant values, and assume that the architecture provides only add and multiply operations. The actual speedup numbers would be much better if either of these assumptions are relaxed.

6 Conclusions

The graphs presented in Section 4 clearly indicate that both symmetric and blocked back-substitution techniques provide performance improvements for machines having some degree of parallelism. This is especially true for first and second order recurrences. Back-substitution techniques can exploit parallelism from either multiple or pipelined functional units.

Symmetric back-substitution may provide better performance if the order of recurrence is high and the machine has a low degree of parallelism. The penalty for back-substituting too far with symmetric back-substitution is greatly reduced performance due to rapid growth in operation count.

The blocked back-substitution technique as defined in this paper has an inherent limitation in that it requires b to be greater than m (the order of recurrence). This makes it inapplicable in the context of higher order recurrences and machines with low degree of parallelism where optimal value of b is quite small. In all other cases, the blocked back-substitution provides better performance improvement. Further work is needed to see if blocked back-substitution offers any advantages over symmetric back-substitution when $b < m$.

When it is useful to expose significant parallelism using a high degree of back-substitution, blocked back-substitution has significantly lower operation count than symmetric back-substitution. Blocked back-substitution allows the exposure of an arbitrary degree of parallelism from a recurrence with no more than a constant multiplier in operation count. The number of operations required for blocked back-substitution does not increase significantly with increased degree of back-substitution and, because of this, one can choose a degree of back-substitution somewhat higher than optimal with little penalty.

This work can be generalized to treat loops with conditional recurrences and loops with exits. In the context of processors that support predicated execution, if-conversion can be used to transform branching code to an algebraic expression. Such expressions can be accelerated using techniques similar to those presented within this paper and will be explored in future work.

Acknowledgments

This work has been influenced by conversations with Bob Rau, Ross Towle, and others who contributed to the Cydra 5 compiler. Work with Robbe Walstra at Hewlett Packard Laboratories helped indicate the weakness of symmetric back-substitution as a height reduction technique.

References

1. H. S. Stone. Parallel Tridiagonal Equation Solvers. ACM Trans Math. Softw 1, 4 (1975), 289-307.

2. S.-C. Chen and D. J. Kuck. Time and Parallel Processor Bounds for Linear Recurrence Systems. IEEE Transactions on Computers C-24, 7 (1975), 701-717.

3. D. Heller. Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems. SIAM J. Numerical Anal. 13, 4 (1976), 484-496.

4. A. H. Sameh and R. P. Brent. Solving Triangular Systems on a Parallel Computer. SIAM J. Numerical Analysis 14, 6 (1977), 1101-1113.

5. D. J. Kuck. The structure of Computers and Computations. (Wiley, New York, 1978).

6. S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical Parallel Band Triangular System Solvers. ACM Transactions on Mathematical Software 4, (1978), 270-277.

7. H. H. Wang. A Parallel Method for Tridiagonal Equations. ACM Transactions on Mathematical Software 7, 2 (1981), 170-183.

8. H. A. Van Der Vorst and K. Dekker. Vectorization of Linear Recurrence Relations. SIAM Journal on Scientific and Statistical Computing 10, 1 (1989), 27-35.

9. Y. Tanaka, *et al.* Compiling Techniques for First-Order Linear Recurrences on a Vector Computer. Proceedings of the Supercomputing Conference (Orlando, FL., 1988), 174-180.

10. D. Callahan. Recognizing and Parallelizing Bounded Recurrences. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Processing (Santa Clara, CA, 1991).

11. J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers C-30, (1981), 478-490.

12. G. Lowney, *et al.* The Multiflow Trace Scheduling Compiler. The Journal of Supecomputing 7, 1/2 (1993), 51-142.

13. W.-M. W. Hwu, *et al.* The Superblock: An Effective Technique for VLIW and Superscalar Compilation. The Journal of Supercomputing 7, 1/2 (1993), 229-248.

14. B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. Proceedings of the Fourteenth Annual Workshop on Microporgramming (1981), 183-198.

15. M. S.-L. Lam, A Systolic Array Optimizing Compiler. 1987, Carnegie Mellon University:

16. B. Rau. Data Flow and Dependence Analysis for Instruction Level Parallelism. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing (Santa Clara, CA, 1992), 236-250.

17. B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code Generation Schemas for Modulo Scheduled DO-Loops and WHILE-Loops. Proceedings of the 25th Annual International Symposium on Microarchitecture (Portland, Oregon, 1992), 158-169.

18. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing 7, 1/2 (1993), 181-227.