Delay Optimization of Carry-Skip Adders and Block Carry-Lookahead Adders

Pak K. Chan, Martine D.F. Schlag*
Computer Engineering
University of California, Santa Cruz
Santa Cruz, California 95064

Clark D. Thomborson
Department of Computer Science
University of Minnesota
Duluth, Minnesota 55812

Vojin G. Oklobdzija
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

The worst-case carry propagation delays in carry-skip adders and block carry-lookahead adders depend on how the full adders are grouped structurally together into blocks as well as the number of levels.

We report a multidimensional dynamic programming paradigm for configuring these two adders to attain minimum latency. Previous methods are applicable only to very limited delay models that do not guarantee a minimum latency configuration. Under our delay model, critical path delay is calculated not only taking into account the intrinsic gate delays, but also the fanin and fanout contributions.

1 Introduction

The worst-case carry propagation delays in carry-skip adders depend on how the full adders are grouped together (into blocks). The problem of configuring carry-skip adders to minimize the carry propagation delay has been the subject of several papers. Lehman has shown that carry-skip adders with variable-size blocks are faster than adders with fixed-size blocks [1]. Later, Majerski suggested that multilevel implementation of the variable-block-size carry-skip adders would provide further improvement in speed [2]. The optimization technique developed for the choice of block sizes by Majerski is limited to a specific ratio between the carry-generate and carry-skip propagation delay. Almost two decades later, Oklobdzija and Barnes developed algorithms for determining *near-optimal* block sizes for one-level and two-level implementations, and a generalization of their method was given by Guyot *et al* [3, 4]. Their algorithms have very elegant geometrical interpretations but do not guarantee optimality of the design. Moreover, their algorithms work only if the carry-skip propagation delay is a constant. In [5] it is noted that in CMOS Manchester adders with carry-skip, the carry-skip propagation delay is not necessarily a constant, but depends on the number of bits in the adder. Chan and Schlag developed a polynomial time algorithm to configure block sizes to attain minimum latency for one-level carry-skip adders under a linear carry-skip delay model. Simultaneously, an indirect enumeration approach was taken by Turrini to generate (multilevel) block distributions containing the maximum number of bits under a specified delay constraint [6]. Unfortunately, this approach is applicable only to constant carry-skip delay models, since it constructs a configuration from the top down by calculating delay constraints for lower-level blocks without knowing the number of bits encompassed in these blocks; when the lowest level is reached each block is filled with the maximum number of bits satisfying its delay constraints.

The idea of varying block sizes to further reduce delays was also suggested in [7], where an exhaustive search was employed to search for an optimum block carry-lookahead adder. Much earlier, Montoye and Cook used an analytical delay model to guide an iterative search for area-time optimal parallel prefix adders generated by a binary recursion [8]. They supplied no runtime analysis of their search technique, although they did indicate that an optimal 34-bit adder could be found in 30 minutes of IBM 3033 time. Wei and Thomborson [9] devised a dynamic programming technique that found, in $O(n^2h^2)$ time, *all* area-time optimal parallel-prefix adders in a class generated by a binary recursion similar to Montoye's. Here, h is the height of the minimum-delay adder of data width n. They found optimal 66-bit adders in a few seconds of SUN-3 CPU time.

In this paper, we formulate the problems of configuring carry-skip adders and variable-block-size block carry-lookahead adders as dynamic programs. The resulting dynamic programs have multidimensional objective functions. It is thus necessary to carry forward a list of optimal structures from each stage of the dynamic program. In the traditional (unidimensional)

*Supported in part by NSF Presidential Young Investigator Grant MIP-8896276

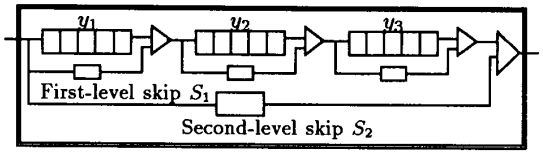

Figure 1: Forming a stage from blocks

dynamic program, only a single optimum structure is carried forward. The existence of multidimensional dynamic programs was noted in an early paper by Dantzig [10]. Weingartner [11] was apparently the first to suggest that this would be an effective method of solving multidimensional knapsack problems. Subsequent researchers [12, 13, 14] refined Weingartner's algorithm, adding more sophisticated data structures and list-pruning strategies.

The multidimensional dynamic programming formulations to configure carry-skip and block carry-lookahead adders, in this article have not been reported elsewhere in form, scope, or generality [1]. In contrast to previously-published optimization techniques, our method immediately generalizes to a wide class of gate delay models and is guaranteed to find minimum latency circuits.

2 Carry-skip adders

2.1 Constructing a stage from blocks

We group several full adders together to form an adder block. Each block has a block-level carry skip mechanism S_1, which can be implemented with a multiplexor selected by the group propagate. The basic structure of a *stage* of a 2-level carry-skip adder is illustrated in Fig. 1. Each stage encompasses several blocks, and contains a second-level carry skip mechanism. The pertinent components of carry-propagation delays in a block are shown in Fig. 2.

2.2 Glossary of terms

The basic notations used in this section are listed below. The meanings of the notations are illustrated in Figs. 1 and 2 .

1. $I(y)$ — internal-carry delay, the maximum delay it takes a carry to generate within a block of y full-adder units and assimilate within the block.
2. $G(y)$ — carry-generate delay, the maximum delay it takes a carry to generate within a block of y full-adder units. This also includes the time it takes a carry to propagate through the buffer (the triangle). Typically, the buffer computes the logical 'or' of its two carry-input signals.
3. $A(y)$ — carry-assimilate delay, the maximum delay it takes a carry to enter a block of y full-adders and assimilate within the block.
4. $S_l(y)$ — lth level carry-skip delay, the time it takes a carry to skip through y full-adder units using the lth-level carry skip mechanism. For $l = 0$, this is the time for a carry to propagate through a block consisting of y full-adder units. For $l \geq 1$, this is the time to compute the logical 'and' of a carry-in signal with the skip-enable signal of the block. This also includes the time it takes a carry to propagate through the buffer.
5. $Set\_up_l(y)$ — lth level setup time, the amount of time it takes to enable the skip circuitry at level l, see Fig. 3. This reflects the delay to generate a group propagate for y bits ($\Pi_{i=1}^{y} p_i$, where p_i is the carry propagate of the ith full adder).

[1] The difficulty of using the dynamic programming technique to solve optimization problems, as noted by Dreyfus [15], lies in the formulation.

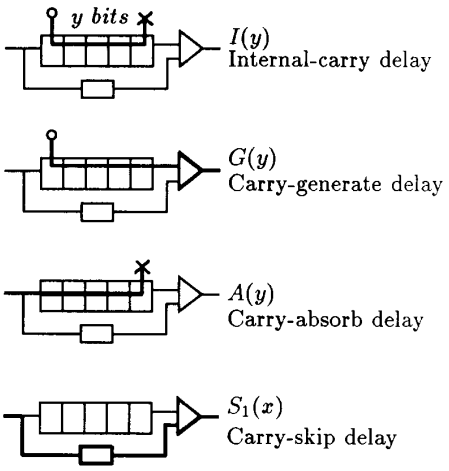

Figure 2: Characterization of delays at the block level

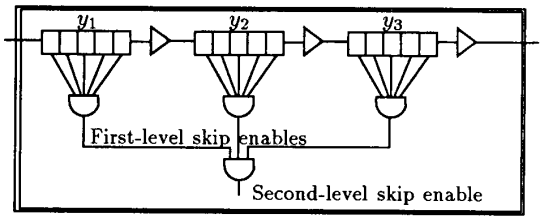

Figure 3: Skip enable generation, inputs to the first level AND gates are the carry propagates p_i

3 A 2-D dynamic programming formulation for finding minimum latency configurations

In order to present the idea in a readily-understandable form, we start the discussion with a two-dimensional optimization problem based on one-level carry-skip adders. The method we derive in this section delivers the same results as a previously-published algorithm [5], but at a much higher computational cost. However, this section's method can be easily generalized to more complicated timing models and to higher-dimensional optimizations.

3.1 Problem statement: one-level carry-skip adder

Let y_k denote the number of bits in block k. We say that a vector $\vec{y} = (y_1, y_2, ..., y_m)$ is an m-block *configuration* of a one-level n-bit adder if $\sum_{i=k}^{m} y_k = n$ and all y_k are positive integers. Let $\mathcal{C}_n$ be the set of all configurations of one-level n-bit adders. We shall assume that all skip circuitries are setup at time zero. The effect of nonzero setup time is treated in Section 3.5. The minimum-latency design problem for a one-level carry-skip adder can now be stated as

> Given timing models for internal-carry $I()$, carry-generate $G()$, carry-assimilate $A()$, and carry-skips $S_0()$ and $S_1()$, find the configuration $\vec{y^*} \in \mathcal{C}_n$ with minimum latency.

The carry-propagation delay between blocks i and j of a configuration $\vec{y}$ is

$$D(\vec{y}, \alpha, \beta) = G(y_\alpha) + \sum_{k=\alpha+1}^{\beta-1} S_1(y_k) + A(y_\beta) \ ; \text{ for } \alpha < \beta \ ,$$

$$D(\vec{y}, \alpha, \alpha) = I(y_\alpha) \ .$$

The worst-case carry-propagation delay of a configuration $\vec{y}$ is therefore $D(\vec{y}) = \max_{1 \le \alpha \le \beta \le m} D(\vec{y}, \alpha, \beta)$. Then our problem is to find a minimum worst-case delay configuration $\vec{y^*}$ for given carry-generate, carry-assimilate and skip delay functions,

$$D_n \equiv D(\vec{y^*}) = \min_{\vec{y} \in \mathcal{C}_n} D(\vec{y}). \qquad (1)$$

3.2 Algorithm: one-level carry-skip adder

We refer to i-bit, j-block carry-skip adders as (i, j)-adders. Note that $j \le i$, since each block must have at least one bit. Also note that, for small blocks, rippling through a single block may be faster than using a one-level skip. For this reason we amend the problem formulation of section 3.1 to allow the possibility of having an initial and/or final block without a skip. During our construction of an optimal configuration we shall consider only (i, j)-adders consisting of an initial (possibly empty) block with no skip, followed by j non-empty consecutive blocks. Given (i, j) there are $\binom{i}{j}$ such adder configurations, since we have the freedom to distribute $i-j$ bits among $j+1$ blocks. A final step will consider adding a block to the end with no skip. We use a pair (t_a, t_g) to characterize the worst-case carry propagation delays of an (i, j)-adder, where

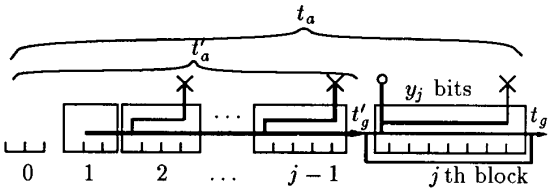

Figure 4: Appending a new block

- t_a is the worst-case delay of any "carry chain" that terminates *before* or *at* block j, and
- t_g is the worst-case delay of any "carry chain" that emerges from block j.

We shall construct for each i and j, a list $t(i, j)$ of pairs (t_a, t_g) for all (i, j)-adders. The basis for the dynamic programming is

$$t(i, 0) = (\max\{A(i), I(i)\}, \max\{G(i), S_0(i)\}); \text{ for } 0 \le i \le n,$$

and for $i \ge j \ge 1$, $i-j+1 \ge y_j \ge 1$, the minimal worst-case delays of (i, j)-adders are formed by composing $(i - y_j, j - 1)$ adders and a new jth block with y_j bits. For each (t'_a, t'_g) in $t(i - y_j, j - 1)$, we construct a pair (t_a, t_g) by:

$$t_a = \max\{t'_a, t'_g + A(y_j), I(y_j)\} \qquad (2)$$

$$t_g = \max\{G(y_j), t'_g + S_1(y_j)\} \qquad (3)$$

We then solve for $D_n = \min_{\vec{y} \in \mathcal{C}_n} D(\vec{y})$ by 2-D dynamic programming in a tableau that retains, for each $t(i, j)$, a list of the minimal (t_a, t_g) pairs for all (i, j)-adders. The list in tableau cell (i, j), for $j \ge 1$, is obtained by using the recursion above to process the lists in cells $(i - y_j, j - 1)$ for all "last block" sizes $i - j + 1 \ge y_j \ge 1$. Once the entire tableau for $1 \le i \le n$ and $0 \le j \le i+1$ has been computed, the lists in column i are concatenated into one list $T(i)$, and a final block of $n - i$ bits without a skip is added. D_n is the minimum of the set

$$\cup_{i=0}^{n} \{\max(t_a, t_g + A(n-i), G(n-i), t_g + S_0(n-i)) | (t_a, t_g) \in T(i)\}.$$

This algorithm delivers the correct minimum for any non-negative $G()$, $A()$, $S_0()$ and $S_1()$ functions, but it potentially requires exponential time and space. The next section addresses this issue by presenting techniques to prune the search and limit the number of configurations generated.

There is a reason to expect good performance, however. If the t_a and t_g values in the retained lists are independently distributed, then each list will have $O(\log n)$ elements with high probability [16, 17]. In this case, the optimization algorithm for an n-bit adder will run in $O(n^3 \log^3 n)$ time, with high probability. Positive correlation among the t_a and t_g values would shorten the lists and hence the runtimes; any negative correlation would lengthen the lists. We expect to see a slight positive correlation for any reasonable delay model, so we believe this method will prove feasible for optimizing adders with hundreds of bits.

3.3 Number of configurations in the tableau

The maximum number of configurations for cell (i, j) in the tableau is the binomial coefficient $\binom{i}{j}$. Fig. 5 shows the potential number of configurations for any 10-bit carry-skip adder. There are ten possible configurations for a 10-bit one-block adder because of the possibility of an initial block (0^{th} block) with no skip. This initial block can hold zero to nine bits. Fortunately, many configurations can be thrown away using the following pruning techniques.

- In each tableau cell $t(i, j)$ the non-dominated pairs must be retained. For example, let (t_a, t_g) and (t'_a, t'_g) be pairs in cell $t(i, j)$ of the tableau. We say (t_a, t_g) is dominated if either ($t_a > t'_a$ and $t_g \geq t'_g$) or ($t_a \geq t'_a$ and $t_g > t'_g$). From equations (2) and (3) it is clear that any pair constructed by adding another block to (t_a, t_g) will be sub-optimal to the corresponding pair generated by adding the same block to (t'_a, t'_g), and hence the former can be discarded. If only one optimal configuration is desired, further pruning can be achieved by breaking ties arbitrarily and discarding all but one of the pairs involved (pruning by domination [11]).
- Once the $t(n, j)$ cell is filled in we can examine its entries and determine the minimum worst-case delay of an (n, j)-adder. This delay, $D_n(j)$, is an upper bound on the final delay, D_n, and can be used to discard any pair generated with either $t_a \geq D_n(j)$ or $t_g \geq D_n(j)$. In order to take full advantage of this bound, we fill each row of the tableau from right to left so that $D_n(j)$ can be used in filling the rest of the row (pruning by fathoming [13]).
- Finally, it is not necessary to fill in the entire tableau. If each pair in $t(i, j)$ is sub-optimal or equal to a pair in $t(i, j-1)$ then $j-1$ is the optimal number of blocks for i bits. There is no point in computing column i above row $j-1$.

Fig. 5 shows the potential number of configurations for any 10-bit carry-skip adder. However, upon using the aforementioned pruning techniques the number of configurations can be drastically reduced. Fig. 6 shows the actual number of configurations generated in each cell of the tableau for the delay model

$$G(y) = y \quad A(y) = y \quad S_0(y) = y \quad S_1(y) = 1.$$

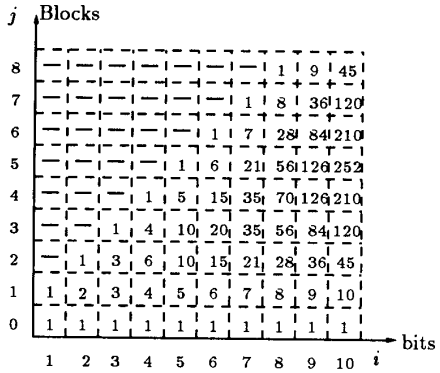

Figure 5: Potential # of entries in the tableau

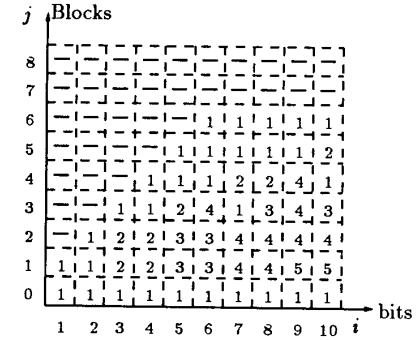

Figure 6: Actual # of entries in the tableau

3.4 Algorithm: l-level carry-skip adder

In this section we generalize the one-level skip algorithm to multiple levels. We shall assume that all skip circuitries are setup at time zero. The effect of nonzero setup time is treated in Section 3.5.

We shall construct carry-skip adders having a total of i bits and j 'stages' *at* level l and denote these as (i, j, l)-*adders*. Again we shall consider only (i, j, l)-adders where the j non-empty stages are consecutive and follow an initial number of bits (possibly none) forming an adder with only lower-level skips. If we were going to apply the algorithm from the one-level case, we would need to have available, $G_{l-1}(y)$, $A_{l-1}(y)$, and $I_{l-1}(y)$ functions. Unfortunately, these

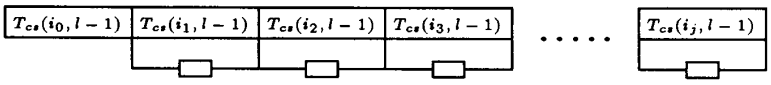

Figure 7: A $(\sum i_k, j, l)$ adder

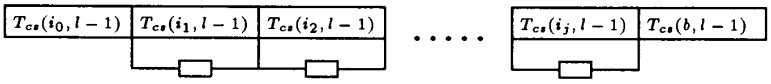

Figure 8: A $(b + \sum i_k, j, l)$ adder

delays are configuration-sensitive and cannot solely be characterized by y. This difficulty is surmounted by determining the values of these delays for all $(i, j, l-1)$-adders. We use a 4-tuple (t_i, t_g, t_a, t_s) to characterize the worst-case carry propagation delays of an (i, j, l)-adder, where

- t_i is the worst-case delay of any "carry chain" that generates *at* or *before* stage j and terminates *at* or *before* stage j (at level l),
- t_g is the worst-case delay of any "carry chain" that generates *at* or *before* stage j and continues through stage j,
- t_a is the worst-case delay of any "carry chain" that enters the adder and terminates *at* or *before* stage j,
- t_s is the worst-case delay of any "carry chain" that enters the adder and continues through stage j.

Again, we shall compute a tableau in which $t_{cs}(i, j, l)$ contains the minimal 4-tuples for all (i, j, l)-adders. Fig. 7 shows a $(\sum i_k, j, l)$-adder. We also characterize the worst-case delays of a 'stage' of a carry-skip adder having i number of bits and l levels *regardless* of the number of stages it contains, with a (possibly zero) number of bits in lower-level blocks at the end. We call these $(i, *, l)$-*adders*, as illustrated in Fig. 8. Their 4-tuples can be obtained from those of the (i, j, l)-adders using the following equations:

$$\begin{aligned} T_{cs}(i,l) \quad &= \{(T_i, T_g, T_a, T_s) \mid \exists\, b, 0 \le b \le i, \\ &\exists\, (T_i', T_g', T_a', T_s') \in T_{cs}(b, l-1), \\ &\exists\, j, \exists\, (t_i, t_g, t_a, t_s) \in t_{cs}(i-b, j, l), \\ &\text{such that} \\ &T_i = \max(t_i, T_i', t_g + T_a') \\ &T_g = \max(t_g + T_s', T_g') \\ &T_a = \max(t_a, t_s + T_a') \\ &T_s = t_s + T_s'\}. \end{aligned} \tag{4}$$

The recurrence relationship for the set $t_{cs}(i, j, l) = \{(t_i, t_g, t_a, t_s)\}$ is defined below. The recurrence formula (5) expresses the worst-case propagation delays of (i, j, l)-adders composed of $(i-b, j-1, l)$-adders and $(b, *, l-1)$-adders.

$$\begin{aligned} t_{cs}(i,j,l) \quad &= \{(t_i, t_g, t_a, t_s) \mid \exists\, b, 0 \le b \le i, \\ &\exists\, (T_i', T_g', T_a', T_s') \in T_{cs}(b, l-1), \\ &\exists\, (t_i', t_g', t_a', t_s') \in t_{cs}(i-b, j-1, l), \\ &\text{such that} \\ &t_i = \max(t_i', T_i', t_g' + T_a') \\ &t_g = \max(t_g' + S_l(b), T_g') \\ &t_a = \max(t_a', t_s + T_a') \\ &t_s = t_s' + S_l(b)\}. \end{aligned} \tag{5}$$

The basis for the dynamic programming is $(l = 0)$

$$T_{cs}(i, 0) = \{(I(i), G(i), A(i), S_0(i))\}\ ;\ \text{for } 0 \le i \le n,$$

and for $l \ge 1$, $t_{cs}(i, 0, l)$ is defined as

$$t_{cs}(i, 0, l) = T_{cs}(i, l-1); \qquad \text{for } 0 \le i \le n.$$

In this expression, the worst-case delay of an n-bit adder using at most l-level skips is the minimum T_i appearing in the sets $T_{cs}(n, k)$, for $1 \le k \le l$.

We control the number of configurations in each set by adopting pruning techniques similar to those described in the previous section. In addition, once an additional skip-level produces only sub-optimal 4-tuples for a given number of bits i, no more new skip levels are considered for i bits.

3.5 Incorporation of setup time for the skip gates

The setup time $Set\_up_l(y)$ is the amount of time needed to enable the skip circuitry at level l. This reflects the delay to generate a group propagate of y bits. Our dynamic programming formulation cannot be easily adapted to take care of the effects of setup time. The problem is that the worst-case absorb and skip times computed for $(i, *, l)$-adders can no longer be used in generating $(i, j, l+1)$-adders since the setup times have been incorporated assuming that carries arrive to the adder at time 0. A compromise is to charge the setup time only to the carry generate,

subsequently the formulation is modified as:

$$
\begin{aligned}
t_{cs}(i,j,l) \quad &= \{(t_i, t_g, t_a, t_s) \mid \exists\, b, 0 \le b \le i, \\
&\exists\, (T_i', T_g', T_a', T_s') \in T_{cs}(b, l-1), \\
&\exists\, (t_i', t_g', t_a', t_s') \in t_{cs}(i-b, j-1, l), \\
&\text{such that} \\
&t_i = \max(t_i', T_i', t_g' + T_a') \\
&t_g = \max(\max(t_g', Set\_up_l(b)) + S_l(b), T_g') \\
&t_a = \max(t_a', t_s + T_a') \\
&t_s = t_s' + S_l(b)\}. \qquad (6)
\end{aligned}
$$

In this formulation, the generate delay t_g is exact, while the other three components may be under-estimated. Care should be taken during the pruning to verify the actual delays of the current best delay for an $(n, *, l)$-adder which will be used to discard configurations.

3.6 Results

We coded our dynamic programming formulation in the "T" programming language [18], and used Turrini's [6] delay model and results to validate our algorithm. Turrini's delay model is

$$
\begin{array}{lll}
I(y) = y & G(y) = y + 1 & A(y) = y - 1 \\
S_l(y) = 1 & Set\_up_l(y) = l + 1. &
\end{array}
$$

Despite the under-estimation of the delay resulting from ignoring the setup time for skips in all but the generate delays, our algorithm was able to generate the same optimal size adders as Turrini [6]. However, we emphasize that our approach is applicable to *any* delay model. Turrini's analysis is limited to models with a constant value for the skip delay, regardless of the number of blocks being skipped.

4 Delay optimization of block carry-lookahead adders

This work was motivated by Wei and Thomborson [9] who used dynamic programming techniques to optimize parallel-prefix adders, as well as a study carried out by Lee [7]. In his paper, Lee discusses the possibility of varying the block sizes in a block carry-lookahead adder (**BCLA**) to further optimize the carry propagation delay. We begin by recalling the structure of block carry-lookahead adders.

Fig. 9 shows a 16-bit 2-level equal-block-size block carry-lookahead adder. Each box is a 4-bit carry-lookahead generator as shown in Fig. 10. These two figures illustrate the notation that we shall be using in this section. We use small letters to denote *global* signal names, e.g., g_0, c_1, and capital letters to denote signal names relative to a block, e.g., G_0, C_1. The goal of our optimization is to minimize the worst-case

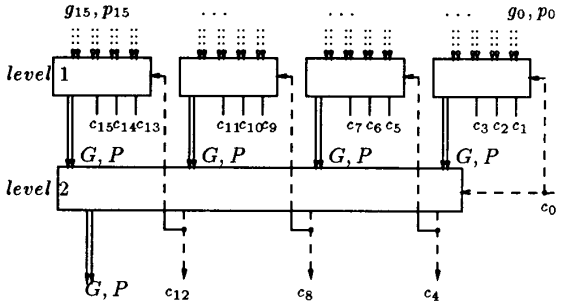

Figure 9: A 16-bit two-level equal-block-size BCLA

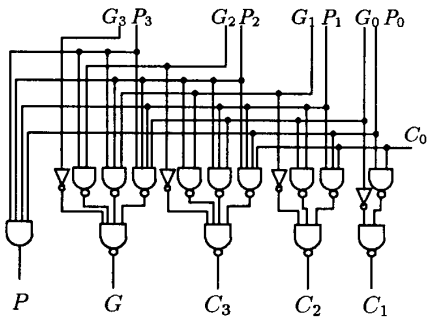

Figure 10: A 4-bit carry-lookahead generator

delay of carries c_1 to c_{n-1} of an n-bit adder. Notice in Fig. 10 that there is no connection from the carry input C_0 of the block to the carry propagate P and generate G outputs. In terms of the structure of the BCLA, this means that P and G are the only signals which travel down the carry-lookahead tree; the carry outputs at the lower levels travel back up to determine the carry outputs of some of their ancestor blocks.

An equal-block-size BCLA minimizes the height of the tree. The latency would be minimal if delays were measured merely by summing unit gate delays along paths. However, in practice the delay of a gate depends on fanin and fanout. The interior of a "block" of a BCLA is a two-level network. Hence the delay of a block is a function primarily of the size of the gates (fanin) as well as the fanout of the signals feeding these gates. Each pair of generate and propagate signals G, P fanout to only one block, however within the block their fanout is linear and quadratic in the size of the block, respectively. These factors tend to limit the block size. In contrast, the carries fanout to multiple blocks (to each of their rightmost ancestor blocks) and hence their delay minimization is improved by decreasing the height of the tree. Smaller blocks are faster and their increased speed may offset additional levels of logic on interior paths, if the sizes of blocks can be varied to balance path delays.

Fig. 11 shows an 8-bit variable-block-size BCLA. Lee shows that the (3-level, 8-bit) BCLA as shown in Fig. 12(a) has the minimum latency according to a gate delay model which considers fanouts and fanins; the next best adder has the configuration of Fig. 12(b). However, Lee found neither exact algorithms nor heuristics to size a BCLA to attain minimum latency [7].

Here, we formulate a multidimensional dynamic program to solve the problem for a particular class of gate delay models, in which gate delay depends *linearly* on fanout and fanin. We refer to BCLA adders having i bits as *i-adders*. For a given m, we construct a BCLA i-adder by selecting m (smaller) BCLA adders of sizes $i_0, i_1, \ldots i_{m-1}$ respectively and combining them to form an $(i_0 + i_1 + \ldots + i_{m-1})$-adder with an m-bit carry-lookahead generator.

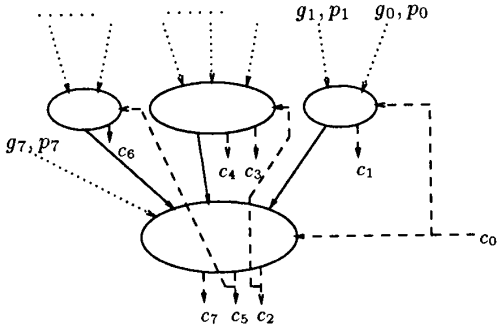

Figure 11: An 8-bit variable-block-size BCLA

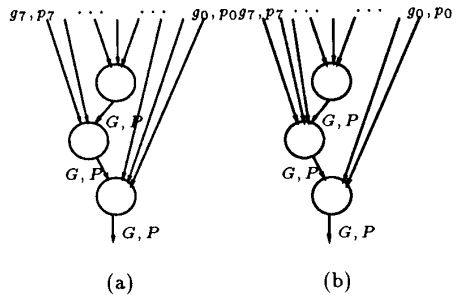

Figure 12: Optimal 8-bit variable-block-size BCLAs

Instead of trying all possible combinations of $i_0, i_1, \ldots i_{m-1}$ which total to i bits, we construct a BCLA adder incrementally, by filling in the positions in our m-bit carry-lookahead generator starting with the least significant position.

A partially completed adder with $i = i_0 + i_1 + \ldots + i_{j-1}$ bits and an m-bit block having only positions $0, 1, \ldots, j-1$ filled is called an (m, i, j)-adder, as shown in Fig. 13. Clearly this is only defined for $1 \leq j < m$. Since not all inputs of the m-bit block are provided, we assume temporarily that these are constants and compute worst-case delays of the adder only from the inputs to positions 0 through $j-1$ of the m-bit block.

Since our goal is to minimize the worst-case carry of an i-adder we must maintain enough delay information in our i-adders and (m, i, j)-adders to compute accurately the worst-case carry delay and guarantee the minimum latency. Because of the structure of the carry-lookahead generator and the gate delay model, we shall be able to compute the delay of an $(m, i, j+1)$-adder without retaining complete information about the arrival time of the inputs to the (m, i, j)-adder. (In fact, the only arrival time that must be retained is that of the most significant generate.) One complication with this construction is that the fanout of the carry-in to an i-adder increases when the i-adder is connected to another block; this may further increase its worst-case carry delay. Fortunately, since the dependence on fanout is linear we can account for the extra delay by maintaining two versions of the worst-case carry delay of an i-adder; one for paths originating from the carry-in and the other for the overall worst-case. Before discussing the delay components which will characterize our i-adders and (m, i, j)-adders in any more detail, we first present our gate delay model.

4.1 Gate delay model

We assume that the input arrival time ($t_{in,j}$) and the output available time (t_{out}) of a gate are related by

$$t_{out} = \max_j \{t_{in,j}\} + FO \cdot \tau + \Delta(FI) \qquad (7)$$

where FO is the fanout of the output signal, FI is the fanin of the gate, τ is the delay per unit fanout, and $\Delta(FI)$ is the delay of a gate of fanin FI under zero load. We define specific delay functions $\Delta_{nand}(FI)$, $\Delta_{and}(FI)$, and Δ_{inv} to model the behavior of 'NAND' gates, 'AND' gates, and inverters under zero load. The functions Δ_{nand}, Δ_{and}, and Δ_{inv} must be monotone nondecreasing, but may take infinite values beyond a certain point in their domain. This ensures that our designs will not contain 17-input-NANDs if an 8-input-NAND is the widest one available.

For simplicity of presentation, we assume that NAND gates, AND gates, and inverters have the same τ value, although this is not a limitation of our formulation. We must, however, require that all Δ functions take nonnegative values over their domains.

We also define δ to express the incremental change of delay per unit fanin: $\delta(FI) = \Delta(FI) - \Delta(FI-1)$. When considering different gates, we add a suffix to identify the gate in question, for example, δ_{and} and

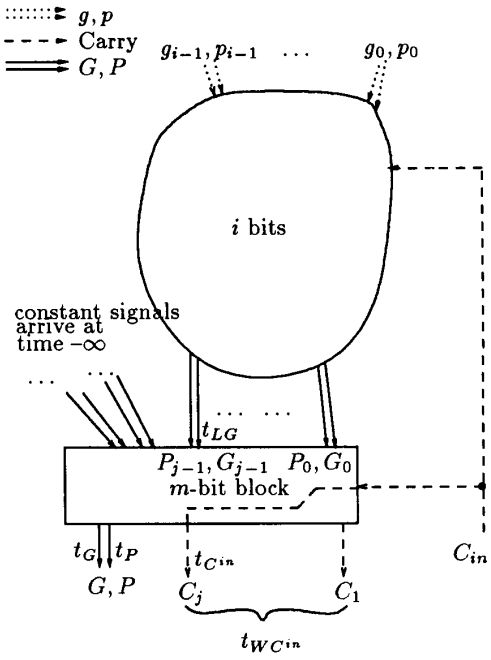

Figure 13: An (m, i, j)-adder

δ_{nand}. Under our linearity assumption on $\Delta()$, δ_{and} and δ_{nand} are nonnegative constants.

The loading on the input signals of a k-bit BCLA adder connected to the jth input of an m-bit BCLA adder [19] can be expressed as

- G_j of the k-bit BCLA adder has fanout $m - j$, i.e., $f_G(m, j) = m - j$. Notice that the fanout is largest at the 0th input position.
- P_j of the k-bit BCLA adder has fanout $f_P(m, j) = (m - j)(j + 1)$.
- The carry-in C_0 to the m-bit carry-lookahead generator has fanout $m - 1$.

4.2 Constructing BCLA adders

For this construction we need to characterize an i-adder with a 5-tuple, $(T_G, T_P, T_{WC}, T_{WC^{in}}, F_{C^{in}})$, where

- T_G is the worst-case delay of the group *generate* output,
- T_P is the worst-case delay of the group *propagate* output,
- $T_{WC^{in}}$ is the worst-case delay of any path from the carry input to any carry output, and
- T_{WC} is the worst-case delay of any carry output,
- $F_{C^{in}}$ is the fanout of the carry input inside the adder.

All the delay values above are calculated under the assumption of zero fanout. When we use these adders as building blocks of larger adders, we shall add appropriate multiples of τ to the delay. Note the two versions of the worst-case carry delay. As discussed, these are necessary in order to account for additional loading on the carry-in when the i-adder is connected to other blocks.

Recall that an (m, i, j)-adder has a partially completed m-bit block with $i = i_0 + i_1 + \ldots + i_{j-1}$ bits having only positions $0, 1, 2, \ldots, j - 1$ filled. We shall characterize an (m, i, j)-adder by an 8-tuple $(t_G, t_P, t_{LG}, t_C, t_{C^{in}}, t_{WC}, t_{WC^{in}}, f_{C^{in}})$, where [2]

- t_G is the worst-case delay of the group generate output,
- t_P is the worst-case delay of the group propagate output,
- t_{LG} is the arrival time of the group generate G_{j-1},
- $t_{C^{in}}$ is the worst-case path delay from the carry input to the currently last carry output C_j of the m-bit carry-lookahead generator,
- t_C is the overall worst-case delay of the currently last carry output C_j of the m-bit carry-lookahead generator,
- $t_{WC^{in}}$ is the worst-case path delay from the carry input to any carry output,
- t_{WC} is the overall worst-case delay of any carry output, and
- $f_{C^{in}}$ is the fanout of the carry input inside the adder.

The arrival time of the input G_{j-1} (t_{LG}) at the m-bit block is the only input arrival time retained. We shall be able to compute all the components of an $(m, i, j + 1)$-adder from those of an $(m, i - b, j)$-adder and a b-adder.

As in the algorithm for carry-skip adders, we retain a tableau of lists for constructed adders:

$$\begin{aligned} T(i) &= \{(T_G, T_P, T_{WC}, T_{WC^{in}}, F_{C^{in}}) \\ &\quad \mid \text{for all } i\text{-adders}\} \\ t(m,i,j) &= \{(t_G, t_P, t_{LG}, t_C, t_{C^{in}}, t_{WC}, t_{WC^{in}}, f_{C^{in}}) \\ &\quad \mid \text{for all } (m,i,j)\text{-adders}\}. \end{aligned} \tag{8}$$

Three sets of equations in our dynamic programming formulation cover respectively, filling in the first position of an m-bit block, an intermediate position, and the last position.

[2] The $t_{C^{in}}$ term is redundant since it will always be equivalent to $f_{C^{in}}\tau + \tau$, but is made explicit here to elucidate the delay equations.

An $(m, i, 1)$-adder is generated from a 5-tuple $(T_G, T_P, T_{WC}, T_{WC^{in}}, F_{C^{in}})$ in $T(i)$, by

$$\begin{aligned}
t_G &= T_G + f_G(m,0)\tau + \Delta_{nand}(m) + \tau + \Delta_{nand}(m) \\
t_P &= T_P + f_P(m,0)\tau + \Delta_{and}(m) \\
t_{LG} &= T_G \\
t_{C^{in}} &= \tau(F_{C^{in}} + m - 1) + \Delta_{nand}(2) + \Delta_{nand}(2) + \tau \\
t_C &= \max\{t_{C^{in}}, \\
&\quad T_G + f_G(m,0)\tau + \Delta_{inv} + \tau + \Delta_{nand}(2), \\
&\quad T_P + f_P(m,0)\tau + \Delta_{nand}(2) + \tau + \Delta_{nand}(2)\} \\
t_{WC^{in}} &= \max\{t_{C^{in}}, T_{WC^{in}} + (m-1)\tau\} \\
t_{WC} &= \max\{t_{WC^{in}}, T_{WC}, t_C\} \\
f_{C^{in}} &= F_{C^{in}} + m - 1.
\end{aligned}$$

For $j > 1$ (and $j < m$), an (m, i, j)-adder is generated by connecting the G_{j-1}, P_{j-1} inputs of an $(m, i-b, j-1)$-adder with the G, P outputs of a b-adder. As discussed, the worst-case carry of the b-adder is adjusted by the increase in fanout of its carry-in. Notice that the two-level network computing the new carry is similar to the network for the previous network; it differs only in that the fanin of the gates have increased and the inputs P_{j-1}, G_{j-1} and G_{j-2} must be incorporated. This is the reason for the retaining the t_{LG} component. As a result, the delay of C_j can be computed based on the arrival times of P_{j-1}, G_{j-1} (provided by the b-adder) and the t_{LG} and t_C components of the $(m, i-b, j-1)$-adder. Specifically, if $(T_G, T_P, T_{WC}, T_{WC^{in}}, F_{C^{in}})$ is the 5-tuple which characterizes the b-adder and $(t'_G, t'_P, t'_{LG}, t'_C, t'_{C^{in}}, t'_{WC}, t'_{WC^{in}}, f'_{C^{in}})$ is the 8-tuple which characterizes the $(m, i-b, j-1)$-adder, then the 8-tuple for our new (m, i, j)-adder is :

$$\begin{aligned}
t_G &= \max\{t'_G, \\
&\quad T_G + f_G(m, j-1)\tau + \Delta_{nand}(m-j-1) + \tau \\
&\quad + \Delta_{nand}(m), \\
&\quad T_P + f_P(m, j-1)\tau + \Delta_{nand}(m) + \tau + \Delta_{nand}(m)\} \\
t_P &= \max\{t'_P, T_P + f_P(m, j-1)\tau + \Delta_{and}(m)\} \\
t_{LG} &= T_G \\
t_{C^{in}} &= t'_{C^{in}} + \delta_{nand} + \delta_{nand} \\
t_C &= \max\{T_G + f_G(m, j-1)\tau + \Delta_{inv} + \tau + \Delta_{nand}(j+1), \\
&\quad T_P + f_P(m, j-1)\tau + \Delta_{nand}(j+1) + \tau + \Delta_{nand}(j+1), \\
&\quad t'_{LG} + f_G(m, j-2)\tau + \Delta_{nand}(2) + \tau + \Delta_{nand}(j+1) \\
&\quad + \delta_{nand} + \delta_{nand},\ t'_C + \delta_{nand} + \delta_{nand}\} \\
t_{WC^{in}} &= \max\{t'_{WC^{in}}, t_{C^{in}}, T_{WC^{in}} + t'_{C^{in}}\} \\
t_{WC} &= \max\{t'_{WC}, T_{WC}, T_{WC^{in}} + t'_C, t_C\} \\
f_{C^{in}} &= f'_{C^{in}}.
\end{aligned}$$

When the mth position of an $(m, i-b, m-1)$-adder is filled in with a b-adder, we obtain an i-adder . If $(T'_G, T'_P, T'_{WC}, T'_{WC^{in}}, F'_{C^{in}})$ is the 5-tuple which characterizes the b-adder and $(t_G, t_P, t_{LG}, t_C, t_{C^{in}}, t_{WC}, t_{WC^{in}}, f_{C^{in}})$ is the 8-tuple which characterizes the $(m, i-b, m-1)$-adder, then the 5-tuple for our new i-adder is :

$$\begin{aligned}
T_G &= \max\{t_G, \\
&\quad T'_G + f_G(m, m-1)\tau + \Delta_{inv} + \tau + \Delta_{nand}(m), \\
&\quad T'_P + f_P(m, m-1)\tau + \Delta_{nand}(m) + \tau \\
&\quad + \Delta_{nand}(m)\} \\
T_P &= \max\{t_P, T'_P + f_P(m, m-1)\tau + \Delta_{nand}(m)\} \\
T_{WC^{in}} &= \max\{t_{WC^{in}}, T'_{WC^{in}} + t_{C^{in}}\} \\
T_{WC} &= \max\{t_{WC}, T'_{WC}, T'_{WC^{in}} + t_C\} \\
F_{C^{in}} &= f_{C^{in}}.
\end{aligned} \tag{9}$$

The basis for the dynamic programming is $T(1) = \{(\Delta_{and}(2), \Delta_{or}(2), 0, 0, 0)\}$. The first two components are the delays to generate the g_i and p_i, respectively.

The minimum worst-case delay of an n-bit BCLA adder is the minimum T_{WC} appearing in the set $T(n)$.

4.3 Implementation

Instead of a 2-D tableau to fill as in the case of the carry-skip adders, we must fill the 3-dimensional volume as depicted in Fig. 14. To construct an (m, i, j)-adder for $j > 1$, we must have already constructed all $(m, b, j-1)$-adders and all b-adders, for $1 \le b < i$. To construct all i-adders we must have constructed all $(m, i-b, m-1)$-adders for $1 \le b < i$ and $2 \le m \le i$. Finally, to construct an $(m, i, 1)$-adder, we must have constructed all i-adders. These dependencies lead us to the following steps depicted in Fig. 14 which are repeated for $z = 2, 3, \ldots, n$.

1. Construct all $(z, b, 1)$-adders for $b = 1, 2, \ldots, z-1$.
2. Construct all (z, b, j)-adders for $j = 2, 3, \ldots, z-1$ and $b = j, j+1, \ldots, z-1$.
3. Construct all (m, z, j)-adders for $m = 2, 3, \ldots, z$ and $j = 2, 3, \ldots, m$. (We now have all z-adders, since they are in fact the (m, z, m)-adders for $m = 2, 3, \ldots, z$.)
4. Construct all $(m, z, 1)$-adders from the z-adders, for $m = 2, 3, \ldots, z$.

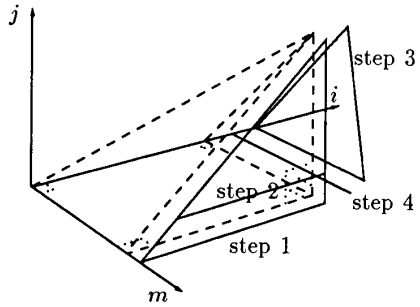

Figure 14: Filling the volume

4.4 Pruning techniques

The maximum number of configurations in the volume even for a small number of bits is prohibitively high. We have employed the following pruning techniques to reduce the number of configurations.

- Set an upper bound on m, the maximum number of inputs to a block, based on a known technological constraint.
- Compute the worst-case delays of equal-block-size adders using the gate delay model. This sets an upper bound on any delay component of the variable-block-size adders that we are building in the "volume." Hence any configuration which has a delay component greater than this upper bound can be thrown away.
- Since the worst-case delays of equal-block-size adders are typically 20% higher than the minimal latency ones, a tighter upper bound can be obtained by temporarily disregarding some delay components during the ranking of the configurations (e.g., t_P or T_P), and running the algorithm to obtain a suboptimal configuration. In effect, this is optimization by a lower-order dynamic program. We then use the maximum delay component of this suboptimal configuration as the new upper bound for a new trial (after reinstating the deleted delay components). This iterative improvement scheme turns out to be very effective in pruning infeasible configurations and reaching the minimum latency configurations.

4.5 Results

We use a gate delay model obtained by fitting data from an ASIC–CMOS standard cell library [20] to linear Δ functions [21]. We select τ to be 5 so that all the parameters in the equations are scaled to integers. Note that an unloaded inverter has delay of 12 units.

$$\begin{aligned} \tau &= 5 \\ t_{NAND,out} &= t_{in} + FO \cdot \tau + FI \cdot 20 \\ t_{OR,out} &= t_{in} + FO \cdot \tau + FI \cdot 20 + 17 \\ t_{AND,out} &= t_{in} + FO \cdot \tau + FI \cdot 20 + 17 \\ t_{INV,out} &= t_{in} + FO \cdot \tau + 12. \end{aligned} \tag{10}$$

We shall represent the carry-lookahead adder tree in parentheses notation. For example, the adder structures shown in Figs. 12a and b are represented as `((1 1 3) 1 1 1)` and `((1 1 1 3) 1 1)`, respectively; and the equal-block-size 16-bit BCLA of Fig. 9 appears as `(4 4 4 4)`. The numbers in the expression represent the block sizes at the top level.

For an n-bit adder, T_{WC} in Table 1 indicates the worst-case delay to generate the carries c_1 to c_{n-1}. The overflow condition is indicated by the final carry c_n which depends on the carry generate and propagate (T_G and T_P). Table 1 shows the delays of variable-block-size BCLAs versus their equal-block-size counterparts. These results are generated by restricting the maximum fan-in of any CMOS gate to 4. The delay of an inverter in a typical 1.5μm CMOS technology is roughly 0.3 ns, so we can convert our integer delay values in Table 1 to nanoseconds in such a technology by multiplying by (0.3/12) ns. For 8-bit adders, we have 9.225 ns and for 16-bit adders we have 12.225 ns.

n	$T_G T_P T_{WC}$	configuration
2	152 124 152	2
3	202 154 202	3
4	309 258 247	((1 2) 1)
	252 184 252	4
6	424 350 314	((1 (1 2)) 1 1)
	337 251 319	(1 1 2 2)
7	399 318 347	((1 1 2 2) 1)
	362 281 364	(1 1 3 2)
8	424 355 369	((1 (1 2) 3) 1)
	382 286 379	(1 1 2 2 2)
	317 251 434	(3 3 2)
	347 251 499	(4 4)
16	559 477 489	((1 (1 (1 2)) (1 1 2) (3 2)) 1 1)
	489 410 506	(((1 2) (1 1 2) 4 (2 1)) 2)
	407 350 569	((2 (2 1)) (2 2 2) (3 2))
	437 311 597	(4 4 4 4)
24	479 385 609	((1 3) (2 (2 2)) (2 3 2) (3 2 2))
	532 378 789	((4 4) (4 4 4 4))
32	621 544 627	(((1 (1 (1 2))) ((1 2) ((1 2) 3)) (2 3 2) (3 2 2)) 2 2)
	532 378 879	((4 4 4 4) (4 4 4 4))
48	716 599 716	(((1 2) ((1 (1 2) (2 2)) ((1 2) (1 2) (3 2)) ((1 2) 3 3 2) (4 3 3))) 3 2)
	577 408 932	((4 4 4 4) (4 4 4 4) (4 4 4 4))
64	789 639 797	((2 3 ((1 2 3 3) (2 3 4 (2 2)) ((1 2 2) (2 2 2) (3 2)) ((2 2 2) (3 2) 3)) (3 2)) 2)
	622 438 977	((4 4 4 4) (4 4 4 4) (4 4 4 4) (4 4 4 4))
66	794 649 802	((2 3 ((1 2 3 3) (2 3 4 (2 2)) ((1 2 2) (2 2 2) (3 2)) ((3 2 2) (3 2) (2 2))) (3 2)) 2)
	717 505 977	(2 ((4 4 4 4) (4 4 4 4) (4 4 4 4) (4 4 4 4)))
84	803 674 856	((3 4 ((2 3 (1 2 2) (3 2 2)) ((1 2 2) (2 3 2) (3 2 2)) ((2 3 2) (3 2 2) (3 2)) ((3 2 2) (3 2) 3)) (3 2)) 2)
	762 530 1069	(4 (4 4 4 4) ((4 4 4 4) (4 4 4 4) (4 4 4 4) (4 4 4 4)))

Table 1: Delays of equal-block-size vs variable-block-size n-bit BCLAs. Configurations for 2-bit to 16-bit adders are optimal (in latency)

Except for the $n \leq 8$ cases, T_{WC} is the dominant delay component. This experiment demonstrates that variable-block-size BCLAs out-perform their equal-block-size counterparts by 15-25%, in terms of their T_{WC}. However, variable-block-size adders are not as modular as the equal-block-size adders. The best variable-block-size BCLAs tend to have more levels but less fan-ins than their equal-block-size counterparts. This suggests that the number of levels is not a good measure of latency for VLSI technology.

5 Conclusion

We have formulated the problems of minimizing the latencies in carry-skip and block carry-lookahead adders as multidimensional dynamic programs. Based on these formulations, we implemented programs to carry out the minimization. The dynamic programming for-

mulations are appealing because of their generality. On the other hand, the computational requirement of the optimization process is also high. All the algorithms presented are coded in the "T" language [18]. The program requires 60 Megabyte of swap space and ran for over 3 hours before completion on a SPARC station. The algorithms generate adder configurations that are not modular, but the adders' latencies are 15-25% less than their modular counterparts.

The delay model that we have established considers fanin and fanout, and is therefore more realistic than counting the number of levels. However, we do not account for the effect of wire lengths in the model, this will be considered in future work.

References

[1] M. Lehman and N. Burla, "Skip Techniques for High-Speed Carry-Propagation in Binary Arithmetic Units," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 691–698, Dec. 1961.

[2] S. Majerski, "On determination of optimal distribution of carry skips in adders," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 45–48, Feb. 1967.

[3] A. Guyot, B. Hochet, and J.-M. Muller, "A Way to Build Efficient Carry-Skip Adders," *IEEE Transactions on Computers*, vol. C-36, pp. 1144–1151, Oct. 1987.

[4] V. G. Oklobdzija and E. R. Barnes, "Some optimal schemes for ALU implementation in VLSI technology," in 7^{th} *Computer Arithmetic Symposium*, pp. 2–8, 1985.

[5] P. K. Chan and M. Schlag, "Analysis and Design of CMOS Manchester Adders with Variable Carry-Skip," *IEEE Transactions on Computers*, vol. C-39, pp. 983–992, Aug. 1990.

[6] S. Turrini, "Optimal group distribution in carry-skip adders," in 9^{th} *Computer Arithmetic Symposium*, (Santa Monica, Los Angeles), pp. 96–103, Sept 1989.

[7] B. Lee, "VLSI Implementation of Fast Arithmetic Algorithms: Optimizing Delays in Carry Lookahead Adders." CS292I: Class project report, UC Berkeley, December 1989.

[8] R. K. Montoye, "Area-time efficient addition in charge based technology," in *ACM IEEE* 18^{th} *Design Automation Conference Proceedings*, pp. 862–872, 1981.

[9] B. W. Wei and C. D. Thompson, "Area-Time Optimal Adder Design," *IEEE Transactions on Computers*, vol. C-39, pp. 666–675, May 1990.

[10] G. Dantzig, "Discrete-Variable Extremum Problems," *Operations Research*, vol. 5, pp. 266–277, 1957.

[11] H. Weingartner, "Capital Budgeting of Interrelated Projects: Survey and Synthesis," *Management Science*, vol. 12, pp. 485–516, Mar. 1966.

[12] H. Weingartner and D. Ness, "Methods for the solution of the multidimensional 0/1 knapsack problem," *Operations Research*, vol. 15, pp. 83–103, 1967.

[13] T. Morin and R. Marsten, "Branch-and-bound strategies for dynamic programming," *Operations Research*, vol. 24, pp. 611–627, July-August 1976.

[14] R. Marsten and T. Morin, "A hybrid approach to discrete mathematical programming," *Mathematical Programming*, vol. 15, no. 1, pp. 21–40, 1978.

[15] S. E. Dreyfus and A. M. Law, *The Art and Theory of Dynamic Programming*. 111 Fifth Avenue, New York, New York 10003: Academic Press, Inc., 1977.

[16] J. Bentley, H. Kung, M. Schkolnick, and C. Thompson, "On the average number of maxima in a set of vectors and applications," *Journal of ACM*, vol. 11, no. 1, pp. 536–543, 1978.

[17] L. Devroye, "A note on finding convex hulls via maximal vectors," *Information Processing Letters*, vol. 11, pp. 53–56, Aug. 1980.

[18] J. A. Rees, N. I. Adams, and J. R. Meehan, *The T Manual*. New Haven, Connecticut: Yale University, March 1983.

[19] T. Rhyne, "Limitations on Carry Lookahead Networks," *IEEE Transactions on Computers*, vol. C-33, pp. 373–373, Apr. 1984.

[20] LSI Logic Corporation, 1551 McCarthy Boulevard, Milpitas, CA 95035, *Compacted Array Technology Data Book*, July 1987.

[21] V. G. Oklobdzija and E. R. Barnes, "On implementing addition in VLSI technology," *Journal of Parallel and Distributed Computing*, vol. 5, 1988.