Adding Faster with Application Specific Early Termination

David Koes Tiberiu Chelcea Charles Onyeama
Seth Copen Goldstein

January 2005

CMU-CS-05-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents a methodology for improving the speed of high-speed adders. As a starting point, a previously proposed method, called "speculative completion," is used in which fast-terminating additions are automatically detected. Unlike the previous design, the method proposed in this paper is able to adapt dynamically to (1) *application-specific* behavior and (2) to *adder-specific* behavior, resulting in a higher detection rate of fast additions and, consequently, a faster average-case speed for addition. Our experimental results show detection rates of over 99%, and adder average-case speed improvements of up to 14.8%.

This research was sponsored in part by the National Science Foundation under grant CCR-0205523 and in part by the Defense Advanced Research Project Agency (DARPA) under contracts N000140110659 01PR07586-00 and MDA972-01-3-0005. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect those of the sponsors.

Keywords: asynchronous hardware, arithmetic and logic structures, high-speed arithmetic, application-specific optimization

1 Introduction

The promise of asynchronous design is based partly on obtaining average case performance from circuits. Synchronous circuits must be designed for worst case performance since the data must always be ready when the clock arrives. Asynchronous circuits, on the other hand, can in principle indicate when the circuit has completed. This promise is often unrealized in the case of datapath elements as the methodology (e.g., dual rail design [19]) used to generate the completion signal incurs significant overhead. Thus, even asynchronous circuits are often designed for worst-case delay. Nowick et al [13, 11] attacked this problem for adders by designing an adder which can complete early, but does not incur the penalty of a full completion detection implementation. We extend their work by developing a methodology which automatically creates specialized early-completion adders. We also redesign the adder circuit such that a static-CMOS design may be used.

As technology continues to scale, increasing numbers of transistors are available to the designer. Using these transistors effectively presents many challenges such as design time, wire delay, and power dissipation. Traditionally, the extra transistors have been used to increase cache sizes, add complicated logic to datapaths, or add extra functionality. An alternative approach is to use the transistors to decrease design time, reduce barriers to further scaling, and lower power. This approach, called spatial computation, distributes the computation over space, i.e., instead of time-multiplexing function units for different parts of a computation, it dedicates a function unit to each operation in a program [7, 18, 5]. This approach reduces energy-delay, decreases reliance on global wires and global controllers, and eliminates many central structures (e.g., register files). A direct result is that every adder in the program is implemented by a separate piece of silicon. Thus, it opens the opportunity to specialize each function unit, in this case each adder, for the context in which it is used.

In order to gain some of the benefit of average case design, Nowick et. al. [11] introduced a modification of the Brent-Kung adder [1] which could detect some cases in which the computation completed before the worst case delay time. The idea is to calculate, in parallel with the addition, a condition which indicates whether the last few levels of the adder are needed. This calculation requires less overhead than traditional completion-detection approaches, but fails to catch all cases in which the adder is done earlier than the worst-case time. They further improve their results by special casing their design, by hand, for different classes of adders (e.g. address arithmetic adders versus general purpose adders).

In this paper we describe our methodology for automatically specializing function units. It is motivated in part by our approach to spatial computation; we transform high-level language programs directly into hardware. It is thus natural to use a profile-based approach to hardware generation. By profiling the program we can determine the statistical nature of the inputs to each adder. We combine the profile data with compiler passes which perform bit-width and constant-bit analyses [4] in order to specialize the adders in two ways. First, the constant-bits and bit-width of the adder is modified to suit the input set. Second, the early-termination detection circuit is optimized so that it will miss the fewest number of cases in which the adder is done early. Because the compiler has access to the distribution of inputs and it creates separate adders for each add in the program, it can create adders specialized to their context without requiring any "by-hand" design.

To validate our methodology, we analyze Mediabench[9], MiBench[8], and Spec[16] benchmark programs. We explore several different algorithms for determining the detection circuit which depend on how the information in the profiled data set is used. Experimental results indicate that, when the number of literals used in early-termination varies from 15 to 35, as many as 96-99% of the early additions are detected, as compared with an average of 76-93% for the Nowick et. al. detection network.

In Section 2 we present some background information including the basic adder used in [13], which is the basis for our design, as well as an overview of “Spatial Computation.” Section 3 introduces our algorithm for building early-termination networks. Section 4 describes the modifications we made to the adder design which reduces the adverse impact of including an early termination path. A detailed evaluation is presented in Section 5, while the paper concludes with a summary and directions for future work.

2 Background

This section reviews some previous contributions which are useful in understanding our approach to building speculative-completion adders. First, techniques for asynchronous completion detection are discussed. Second, the basic Brent-Kung adder is described. Third, the original Nowick speculative completion adder is reviewed. Finally, an overview of spatial computation is provided.

2.1 Completion Detection

Asynchronous circuits must rely on a signal other than the clock to indicate the availability of results. The techniques for generating this signal usually fall into one of two categories: *matched delay* and *completion detection*.

A *matched delay* [6, 2, 14] approach computes the worst-case delay of the functional unit and inserts a delay element which matches this worst-case delay through the functional unit. The delay element can either be an inverter chain, or, for a tighter match, a replication of the critical path through the functional unit. The main advantage of this method is that widely available synchronous implementations for functional units can be used; the disadvantage is that the delay of the functional unit is data-independent and always worst-case.

A *completion detection* method [10, 15] is data-dependent. It uses the result signals, usually encoded using *delay insensitive* codes [19], to detect when the computation is done. However, its main disadvantage is the completion detection network which adds overhead to the delay of the result and may offset the savings of detecting true average-case. In addition, the use of delay-insensitive codes requires the design of non-standard functional units and vastly increases their area.

The methodology used in this paper falls in between these two methods. The method, called “speculative completion” was introduced by Nowick in [13, 11] and is described in detail below.

2.2 Brent-Kung Adder

In [1], Brent and Kung proposed a fast, parallel carry-lookahead adder. This adder is the basis for both Nowick’s speculative completion adder and the adder proposed in this paper.

Figure 3 shows the schematics of a 32-bit Brent-Kung adder. The adder consists of 7 levels of logic, each with a simple CMOS implementation and a fanout of 2. In general, an n-bit adder is implemented using $log(n) + 2$ levels of logic. The adder is fast, amenable to regular layout, and, most important for our work, it is based on a simple, repetitive structure, which can easily be generated automatically.

The Brent-Kung adder is implemented as follows. Level-0 produces all propagate ($\overline{p}$) and generate ($\overline{g}$) signals. Level-6 produces the sum as an XOR of level-0 propagates and level-5 generates. The intermediate levels compute the propagate and generate signals for increasing runs of bits: level-1 computes all 2-bit P and G values, level-2 all 4-bit P’s and G’s, and so on. In general, the expressions for level i and bit j propagates and generates are: $P_j^i = P_j^{i-1} \cdot P_{j-k}^{i-1}$ and $G_j^i = G_j^{i-1} + P_j^{i-1} \cdot G_{j-k}^{i-1}$, where $k = 2^i$.

2.3 Speculative Completion

Speculative Completion [13, 11] is a method for designing asynchronous datapath components. This method has the advantages of matched delay implementations—the use of single-rail synchronous functional units, but, more in the spirit of completion detection implementations, the functional units have several associated delays: one for the worst case, and one or more speculative delays for the cases where the result is ready earlier. The functional unit is also augmented with a termination-logic network, which operates in parallel with the functional unit, and speculatively selects between the several associated delays.

As an exemplification of speculative completion, Nowick et al. have proposed a speculative adder [13, 11]. This adder is based on the Brent-Kung adder. The adder can either complete its computation "early" (after only 4 levels of logic), or "late" (after all 6 levels of logic). They prove that the necessary condition for "late" completion is: *Late completion can only occur if there exists a run of 8 consecutive Level-0 propagate signals.*

The termination-logic network detects when an addition may complete late. Since an exact implementation of late completion detection is prohibitive (the SOP has exactly 200 literals), the termination logic only *safely approximates* the condition for late completion. Nowick has proposed three different termination-logic networks for general 32-bit adders. They also propose several other networks which can be used for specialized adders, for example, adders which are guaranteed to have one summand of only a few bits. However, these termination-logic networks have fixed implementations, and can not adapt to adders with a non-uniform distribution of inputs.

2.4 Spatial Computation

"Spatial Computation" (SC) [5, 3] is a model of computation, which is based on the translation of high-level language programs directly into hardware structures. SC program implementations are completely distributed, with no centralized control. SC circuits are optimized for *wires* at the expense of computation units.

A particular implementation of SC is ASH (Application-Specific Hardware) [5]. Under the assumption that computation is cheaper than communication, ASH replicates computation units to simplify interconnect, building a system which uses very simple, completely dedicated communication channels. As a consequence, communication on the datapath never requires arbitration; the only arbitration required is for accessing memory. ASH relies on very simple hardware primitives and uses no associative structures, no multiported register files, no scheduling logic, no broadcast, and no clocks. As a result, ASH hardware is fast and extremely power efficient.

CASH is a fully automated compiler which takes ANSI C as input and translates it into Pegasus [3], a dataflow intermediate representation. CAB (CASH Asynchronous Back-End) [18] then takes the Pegasus representations and synthesizes them into micropipelined implementations, where each stage communicates using 4-phase bundled-data protocols. CASH performs a wealth of optimizations on the code, of which bit-width and range analysis, and constant propagation are of direct consequence to this paper since they allow us to create specialized adders.

To better understand the role of Spatial Computation in specializing adders, Figure 1 shows the Pegasus representation of a simple sum-of-squares program, that uses `i` as an induction variable and `sum` to accumulate the sum of the squares of `i`. On the right is the program's Pegasus representation, which consists of three hyperblocks [5]. Hyperblock 1 initializes `sum` and `i` to 0. Hyperblock 2 represents the loop; it contains two MERGE nodes, one for each of the loop-carried values, `sum` and `i`. Hyperblock 3 is the function epilog, containing just the RETURN. Back-edges within a hyperblock denote loop-carried values; in this example

```
int squares()
{
  int i = 0,
      sum = 0;

  for (;i<10;i++)
      sum += i*i;
  return sum;
}
```

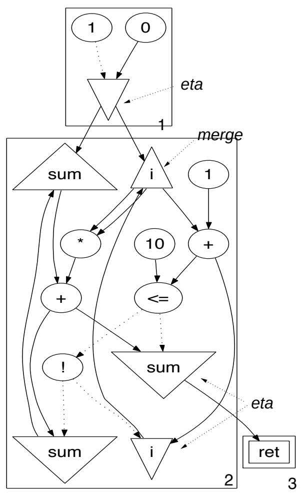

Figure 1: *C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)*

there are two such edges in hyperblock 2; back-edges always connect an ETA (loop exit) to a MERGE (loop entry) node.

Figure 1 has two individual 32-bit adders. However, each of these adders is strictly specialized. The adder for the induction variable always adds with 1, whereas the adder for the accumulation of sum adds more variable values. This information, easily obtained by analysis in CASH, helps us to specialize the termination-logic network for each adder. For example, for the addition with constant ‘1’, this network can be as small as a single literal (p_7): the upper 31 bits of constant ‘1’ are all zero’s, which means that the only time a late addition occurs is when there is a run of propagate bits originating in position 1 and ending in position 7. Of course, for a more accurate detection, more literals should be used. Note, however, that they need not be evenly distributed throughout the 32 bits and rather they should be concentrated on the lower 8 bits.

3 Early Termination Detection

The early-termination detection network must quickly determine if all the generate signals at a given level are equal to the final result. If so, it is safe to terminate the addition early. The generate signal for bit i at level n, G_i^n, is the sum $G_i^{n-1} + P_i G_{i-2^{n-1}}^{n-1}$. Clearly, if $P_i = 0$ then $G_i^n = G_i^{n-1}$. This is a sufficient, but not necessary condition, for early completion. $P_i = 0$ if and only if $\prod_{j=i-7}^{j=i}(p_j = 0)$. If all such products are zero, then the propagate signals are all zero and the generate signals of successive levels remain unchanged. Therefore, we can predict the early-termination behavior of an addition as a function of the level-0 propagate bits, $\overline{p}$, with the follwing function $LATE$ which evaluates to zero if and only if the addition can be early terminated:

$$LATE = p_0p_1p_2p_3p_4p_5p_6p_7 + p_1p_2p_3p_4p_5p_6p_7p_8 + \dots$$
$$+ p_{24}p_{25}p_{26}p_{27}p_{28}p_{29}p_{30}p_{31}$$

We refer to each term in $LATE$ as a *run* and denote the run starting at bit i as r_i. For example, the first term is run r_0.

It is not feasible for the early-termination detection network to exactly compute $LATE$. Instead, a safe approximation is used. The $LATE$ function is safely approximated if all the runs, $r_0 \ldots r_{24}$, are *covered*. Runs are covered by a function over $\overline{p}$ if the function safely approximates the contribution of each of the runs to $LATE$; that is, if the function will only return zero if the OR of the covered runs is zero (but the function, being an approximation, may also return a one in this case). A safe approximation never incorrectly identifies an addition as being early, but may incorrectly identify an addition as being late.

Previous work [13] has used the following uniform safe approximations in early-termination networks:

- $L_{3\times5} = p_5p_6p_7 + p_{11}p_{12}p_{13} + p_{17}p_{18}p_{19} + p_{23}p_{24}p_{25} + p_{29}p_{30}p_{31}$
- $L_{4\times5} = p_4p_5p_6p_7 + p_9p_{10}p_{11}p_{12} + p_{14}p_{15}p_{16}p_{17} + p_{19}p_{20}p_{21}p_{22} + p_{24}p_{25}p_{26}p_{27}$
- $L_{5\times7} = p_3p_4p_5p_6p_7 + p_7p_8p_9p_{10}p_{11} + p_{11}p_{12}p_{13}p_{14}p_{15} + p_{15}p_{16}p_{17}p_{18}p_{19} + p_{19}p_{20}p_{21}p_{22}p_{23} + p_{23}p_{24}p_{25}p_{26}p_{27} + p_{27}p_{28}p_{29}p_{30}p_{31}$

As the number and size of terms grows, the accuracy of the approximation increases, at the cost of increased circuit size and execution time.

3.1 Using Application Specific Information

Here we describe how we exploit application specific information to design early terminating adders which exploit application specific information. Instead of implementing a uniform function in the early-termination network, a custom function is used that, given the runtime behavior of the application, more accurately approximates the $LATE$ function while not exceeding timing and space constraints.

We use dynamic programming to generate the custom approximation function. The dynamic programming algorithm builds up a solution by finding the function F_{ij} with the best score B_{ij} which covers the runs $r_i \ldots r_{last}$ using no more than j literals. The score of a function is determined by a *probability predictor* which, based on profile information, returns the probability that the function returns zero.

A safe approximation of $LATE$ can be built up from literals and terms. A literal p_i covers a run r_j if p_i is contained within r_j. For example, p_1 covers both the runs r_0 and r_1. If $p_i = 0$, r_j must be zero whereas if $p_i = 1$ we can safely approximate r_j as being one. A run is covered by a term if it is in the intersection of the runs covered by the literals of the term. For example, the term p_1p_8 covers only the run r_1. A disjunction of terms covers the union of the runs covered by the terms of the disjunction. For example, $p_7 + p_{15}$ covers the runs $r_0 \ldots r_{15}$.

Our algorithm for building up a termination approximation function from profile data is given in Figure 2. The algorithm builds up a safe approximation function by greedily adding a term that covers an additional run to a previously computed function such that the probability that the resulting function returns zero is maximized. The algorithm is not optimal since the term and bit probabilities are not independent. If no independence assumptions were made, an optimal algorithm would have to consider an exponential number of functions. The running time of the algorithm is $O(NL\Pi R2^R)$ where N is the number of bits, L is the maximum number of literals desired, Π is the running time of the probability predictor, and R is the number of bits in a run. As N, L, and R are typically small constants, the running time of the algorithm is dominated by the probability predictor.

```
for $i = MAX\_RUN$ to $0$ do
  for $l = 1$ to $L$ do
    $best\_prob = -1$
    $best\_func = \emptyset$
    for all $t$ such that $t$ covers $r_i$ do
      $l' = l - num\_literals(t)$
      for all $j > i$ s.t. $\forall_{k \geq i} r_k$ is covered by $F[j][l'] + t$ do
        if $P(F[j][l'] + t) > best\_prob$ then
          $best\_prob = P(F[j][l'] + t)$
          $best\_func = F[j][l'] + t$
        end if
      end for
    end for
    $F[i][l] = best\_func$
  end for
end for
```

Figure 2: Dynamic programming algorithm for building a function with at most L literals that safely approximates the $LATE$ function and maximizes the probability that the function returns zero (early) as determined by a probability predictor P modulo the (false) assumption that the probability of a term being zero is independent of the value of other terms. For clarity the boundary condition checking is ommitted.

3.2 Probability Predictors

Both the executation time and result quality of our termination approximation function generator depend heavily on the probability predictor. The probability predictor uses profile data to assign probabilities that a given function of $\overline{p}$ is zero. The profile data consists of a trace of addition inputs for some execution of a program. A perfectly accurate probability predictor would evaluate the function on every addition in the trace and return the percentage that evaluated to zero. Such a predictor would run in time proportional to the size of the trace file and so would not be feasible in practice. Instead, it is necessary to summarize the data and extract an approximate probability from the summary. We consider three different probability predictors with different trade-offs between accuracy and running times:

Bits Predictor This predictor summarizes the data using bit frequencies. The probability of a function is determined by assuming bit independence. The summary of the profile data can be collected and stored in $O(N)$ space, but the predictor makes strong independence assumptions that affect its accuracy. The running time of this predictor is $O(NR)$.

Terms Predictor This predictor summarizes the data using term frequencies. The probability of each term at every bit position is stored. The probability of a function is determined by assuming indepence between term probabilities. The summary of the profile data can be collected and stored in $O(N2^R)$ space, but the predictor makes weaker independence assumption than the bits predictor. The running time of this predictor is $O(N)$.

Sampling Predictor This predictor summarizes the data by randomly selecting a subset of the data. The probability of a function is determined by evaluating the function on all the members of the sample. This predictor requires an arbitrary amount of memory depending on the sample size. The accuracy

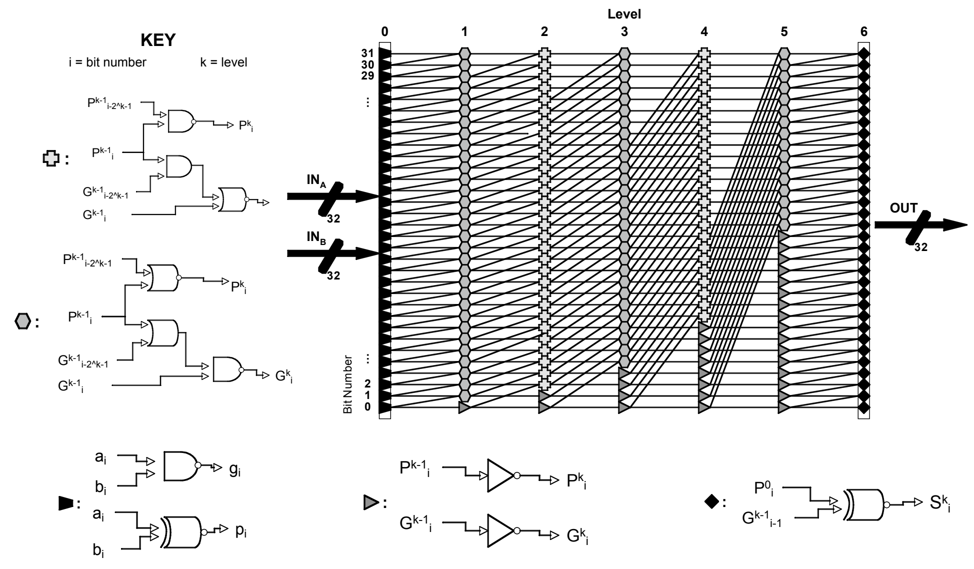

Figure 3: *Brent-Kung adder*

of the predictor is expected to increase with the sample size, but the running time is $O(NRS)$ where S is the sample size.

4 Adder Architecture

The previous two sections have described the basic Nowick speculative completion adder and our algorithm to dynamically construct termination-logic functions. This section describes the modifications to the original Nowick adder. First, we describe the changes in sum generation which render static-CMOS implementations speed-effective. Second, we briefly present several issues involved in the technology mapping of termination-logic networks.

4.1 New Adder Implementation

The original speculative adder in [13] was built using dynamic logic and manually implemented at transistor-level. Although still correct, a static-CMOS implementation has the main disadvantage of introducing significant delays in the sum generation level, resulting in an average-case that is greater than the latency for the unmodified Brent-Kung adder. In our system, CAB synthesizes circuits using standard gates from vendor libraries, so dynamic logic implementations were not available and a better solution than the original was required.

Two small, yet crucial, modifications of the original speculative adder design yield speculative adders useful for standard-gate implementations. The first modification addresses the sum generation logic; the second one modifies the implementation of intermediate “generate” cells.

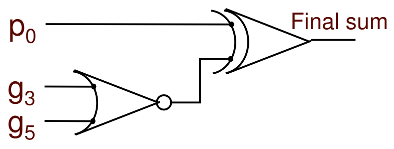

Figure 4: *Modification for sum generation at level-6.*

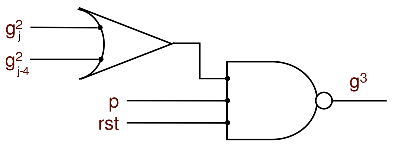

Figure 5: *Generate function at level three.*

4.1.1 Sum Generation

The original sum generation logic [13] (see Figure 3) increases the delay of the last level in the Brent-Kung adder from one XOR gate to four gates, including three XOR gates and one complex AOI gate. As recognized by Nowick, this unacceptably increases the average-case delay of the adder, which now may top the delay of the un-modified adder. Therefore, to render a speculative adder useful in standard gate implementations, a better solution is introduced here.

The sum generation cell for a 32-bit adder implements $s_i = p_j^0 \oplus (early \cdot G_j^3 + \overline{early} G_j^5)$. First, it has to select between the generate signals from level-3 (for early completion) or level-5 (for late completion). Second, it needs to XOR the selected generate signal with the propagate from the first level of logic. A direct implementation would introduce a MUX gate on the critical path, and would require broadcasting the result of termination logic to all sum generation cells.

Our new sum implementation takes into account the nature of the generate signals. The generate function for each level is $G_j^i = G_j^{i-1} + P_j^{i-1} \cdot G_{j-k}^{i-1}$, where i is the adder level, j is the bit position, and $k = 2^{i-1}$. This function is monotonically increasing in the value of G_j. Therefore, assuming the initial values of G_j^3 and G_j^5 are zero, the selection of the proper generate signal in the sum generation is simplified to an OR gate. Figure 4 shows the new implementation for the sum generation; since level-3 and level-5 generate signals are negated, the OR gate becomes a NAND gate.

4.1.2 Partial Reset Logic

The new sum implementation assumes that G_j^3 and G_j^5 are zero at the start of the computation. However, this may not be the case, and the adder needs to be reset before performing a new addition. Such a solution may be un-acceptable from the point of view of performance penalties introduced: either extra gates need to be placed on the inputs, or on the intermediate-level gates.

Our solution to this problem is to reset the adder *partially*, i.e. reset only selected signals inside the adder. The sum generation uses only level-3 and level-5 generate signals, so these are obvious candidates. However, a better solution is possible. Since the generate functions are monotonically increasing, it is sufficient to reset only the level-3 generate; after two gate delays, level-5 generates also becomes zero.

The implementation for the new level-3 generate function is shown in Figure 5. It has only one extra input, a reset signal, which is *active-0*: 1 during computation and 0 during reset.

The new adder introduces a timing constraint on the environment: the delay between two consecutive additions has to be greater than the delay of level-4 and level-5 generate cells combined. In effect, the adder

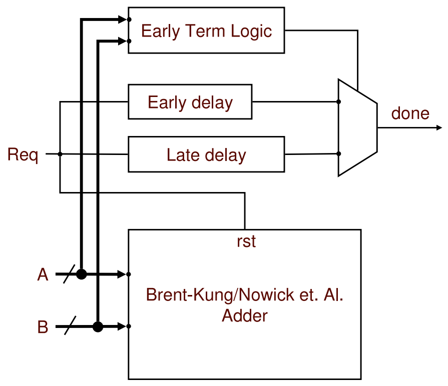

Figure 6: *Architecture of our early terminating adder.*

now works in *fundamental-mode* [17, 12]. However, it is very unlikely that this constraint will be violated, since the adder is assumed to work in designs communicating with 4-phase handshaking: the return-to-zero phase (which include the partial adder reset) is at least the worst-case delay of the adder.

The main drawback of the new adder is increased power consumption since the level-6 sum bits now potentially perform twice the number of transitions of the original adder during each cycle of computation.

4.1.3 Adder Architecture

Figure 6 shows the new architecture of the adder. At this level, the only difference from the original speculative-completion adder is the introduction of a new RESET signal which is used as described above. Since it is assumed that the adder is used in modules which communicate with 4-phase handshaking, the RESET signal can simply be connected to the “req” input: when “req” = 1, the adder is ready to perform its computation, when “req” = 0, the adder is partially reset for the next computation.

4.2 Technology Mapping for Termination Logic

The termination logic for speculative adders has to obey two constraints. First, the delay of the termination logic has to meet a timing constraint to avoid hazards. Second, its output has to be glitch free after a certain delay.

The output of the termination-logic network selects between the early delay and the late delay; the output of the selection is the “done” signal for the entire adder, which has to be glitch-free. The matching delays are built as chains of inverters, so they are hazard-free. Therefore, the termination logic has two constraints. First, it has to compute faster than the early delay ($\delta_{TL} < \delta_{early}$). Second, the termination logic has to maintain a glitch-free output δ_{early} after the start of the computation.

The termination logic is technology-mapped by our compiler. The input is a simple 2-level logic function, and the output is structural Verilog. The compiler applies simple transformations to the logic function (factoring, deMorgan’s laws), producing a fast and small implementation. However, the actual delays of the adder and of the termination logic are not known until later, when commercial CAD tools estimate the actual delays after place and route. Therefore, our solution is to characterize these delays in the back-end, and check for the timing constraint $\delta_{TL} < \delta_{early}$. If the condition holds, then the adder is correct; otherwise, the adder is flagged, and the compiler is run again to generate this particular adder as a standard, non-speculative adder.

5 Results

We evaluate our application specific early-terminating adders in the context of spatial computation and Application Specific Hardware (ASH). The optimizing CASH compiler compiles standard C programs into asynchronous circuits represented in structural Verilog. The gate-level Verilog generated is synthesized with Synopsys Design Compiler 2002.05-SP2, placed-and-routed with Silicon Ensemble 5.3, and simulated with Modeltech Modelsim SE5.7.

In the context of our work, the important feature of the spatial computation model is that, unlike a monolithic processor, there is not just a single adder. Instead, every static add in a program translates into an adder in the final circuit. This feature allows us to not only create an application specific custom adder, but to create many application-context specific adders. Previous work[13] considered address computations, branch offsets, and ordinary arithmetic as separate cases; with spatial computation we can take this specialization to the extreme and optimize each adder based not only on the type of the add, but the program context.

A functional simulator of ASH was used to collect traces of additions and subtractions. The complete traces were than summarized as appropriate for each probability predictor and the termination-logic function was generated for each individual adder as well as for a single monolithic adder. The generated functions were then provided as input to the compiler, which generates speculative adders[1].

5.1 Early-Termination Detection

Although our modified adder completes faster than an un-modified one if it terminates early, it is slower than un-modified when it completes late. Thus, in order for early termination to be an effective strategy, there must be a significant percentage of adds which can be detected as being early. In our 32-bit implementation, an early add completes in .56ns, but a late add completes in .73ns compared to .66ns for a normal adder (see Table 1). This means that more than 41.17% of the additions must be early, or the average execution time of the adder will be worse than the standard adder. As shown in Figure 7, early adds are common enough that implementing early termination is beneficial. For subtractions, only 8 out of 31 benchmarks would benefit from using an early-termination mechanism; to make subtractions effective, more application specific information must be used. This is a topic for future research.

The accuracy of our lateness approximation functions is shown in Figure 8. We compare functions with at most 35 literals derived using different probability predictors as well as the monolithic versus spatial models. As expected, the spatial model using the terms predictor does the best with an average error of less than one percent. Using application specific information is useful even under the monolithic model with the bits and terms predictors averaging 5.17% and 4.54% error versus 6.85% for the uniform function.

The advantage of using application specific information is greatest when the number of literals in constrained. An termination-logic network that takes 35 literals as input is still fast enough to successfully trigger early termination (Figure 10). However, using fewer literals results in a smaller circuit and reduced power consumption. As shown in Figure 9, the combination of optimizing for application specific behavior and the spatial computing model results in lateness approximation function which, using only 15 literals, have better accuracy than a uniform function of 35 literals.

Even when a maximum of 35 literals are allowed in the lateness approximation functions, the full benefit of the function can be obtained by using fewer literals. The distribution of function sizes is shown in Figure 11. This distribution is bimodal. Adders where only the lower bits are typically affected, such as

[1]We are currently integrating the speculative adder generation with the rest of the CAB synthesis path. We will have the simulations of Mediabench kernels within one month.

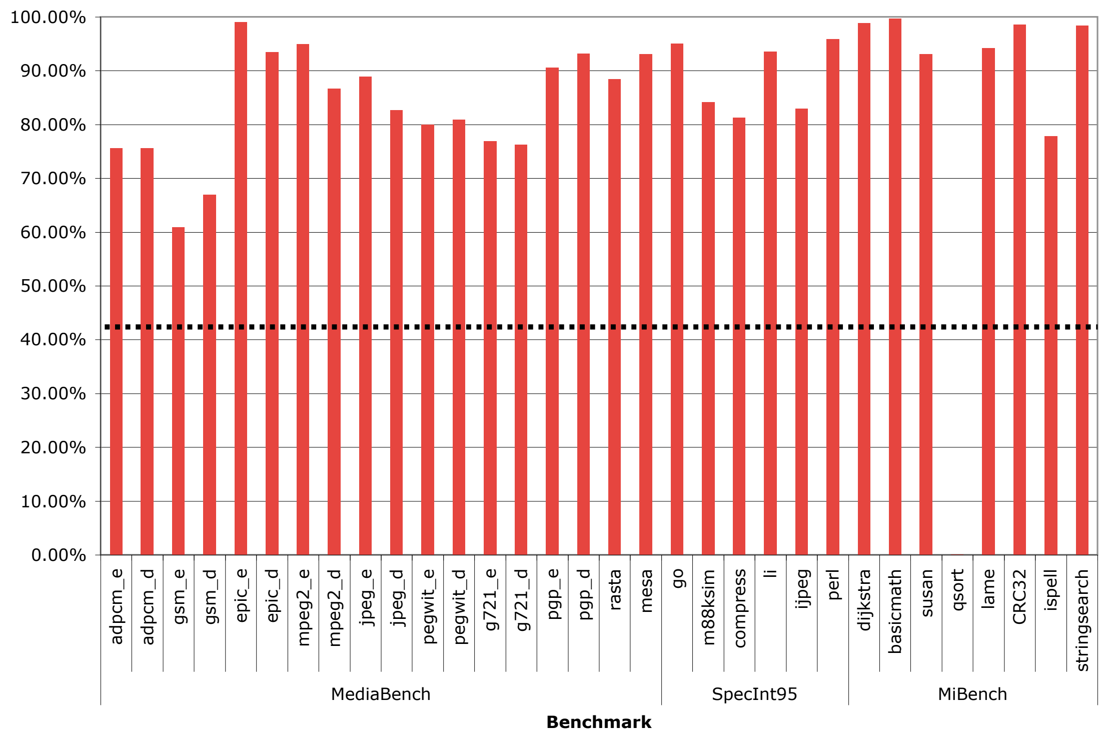

Figure 7: The percent of all additions in each benchmark that support early termination as determined by $LATE$. The threshold above which it becomes beneficial to use an early-terminating adder (the overhead of the implementation is compensated for by the performance gains of the early additions) is shown as a dashed line. All but one benchmark exceed this threshold.

those involved in address computations and incrementing, need fewer than the maximum number of literals to detect their behavior while adders involved in more complicated arithmetic can take full advantage of all available literals.

One possible disadvantage of using a profile driven optimization is that the result of the optimization, in this case the early termination detection functions, may be overly specific to the profiled data and not be representative of the general behavior of the program. To demonstrate this effect, the accuracy of the early termination detection functions was evaluated on programs from Mediabench[2] when run using different inputs and options than the original profiled execution (Figure 12). In many cases it is clear from the significant decrease in the accuracy of early termination detection that the functions are overly specific to the profiled input sets. These results clearly indicate that additional profile data should be used to create the best termination networks for the application. For example, the error of the terms based early termination detection functions for the spatial computation model is over 60% on the benchmark `epic_d` when only the initial profile data is used, but when the combined profile data of both sets of inputs is used to create the funtion the error drops to 1.57% although the error on the original input set increases from .71% to 2.07%. If this new function is then evaluated on another, unprofiled, execution trace the error of this function is

[2]The other benchmarks either did not have alternative inputs or did not have alternative inputs where the addition trace was a manageable size.

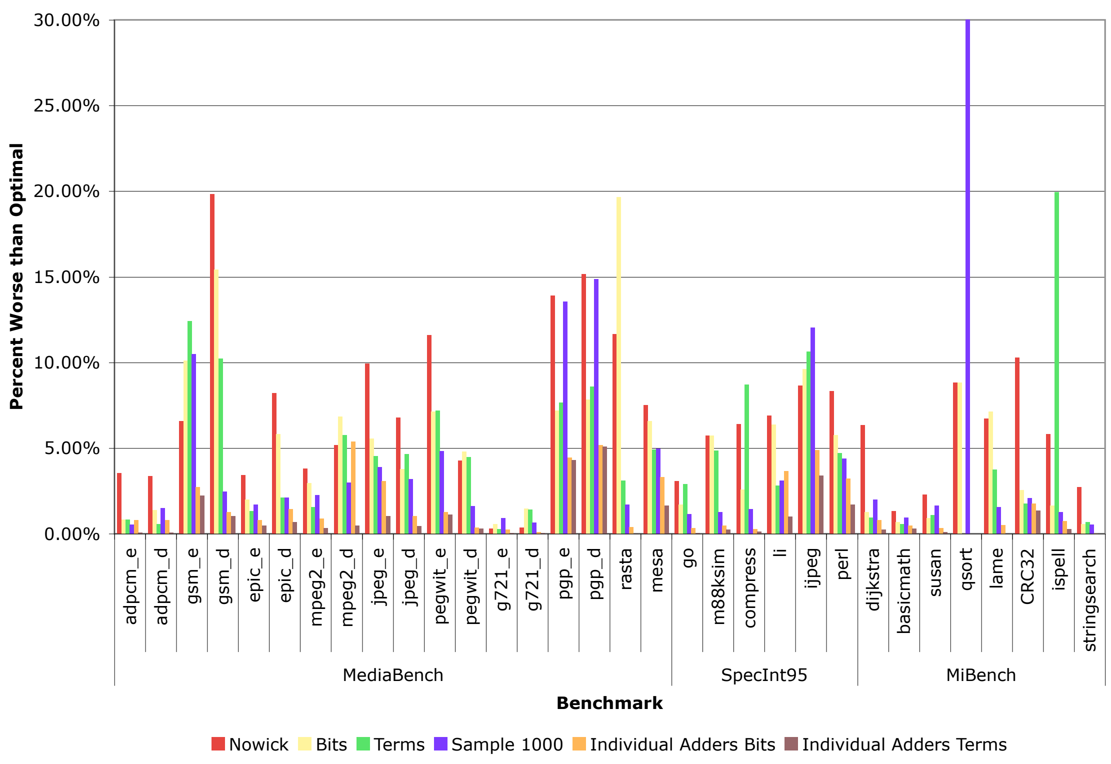

Figure 8: The percent error of various lateness approximation functions relative to the $LATE$ function. As expected, the application specific functions do better than the uniform function, but are eclipsed by the individual adders, where each static add receives its own custom-tailored early termination network. All functions were restricted to containing no more than 35 literals.

4.93% which is comparable to Nowick's function. However, the early termination detection functions for the monolithic model perform significantly better (2.35% for bits and 2.08% for terms).

5.2 Adder Performance

The performance of the dynamically generated adders is investigated in this subsection. First, the termination-logic network is characterized. Then, the dynamically generated adders from the `gsm_d` bench are characterized for speed, area, and power.

The delay of the termination logic *must* be less than the early delay through the adder. Figure 10 shows a graph of the delays through termination-logic functions, as well as the delay threshold for early addition for a 32-bit adder. The termination-logic functions were randomly generated, and the maximum delay through each is shown here. Notice that for termination functions of up to 50 literals, the timing constraint on the termination logic is met. In practice, however, the number of literals in the termination logic is much less than this threshold number, and the delays through the termination logic are much smaller than the early completion delays (see Table 1).

In order to characterize the performance of the dynamically-generated speculative adders, the `gsm_d` benchmark was used as an example. This benchmark generates 7 different adders (Table 1). Of these, two (112 and 173) are general-purpose adders, while the other are specialized adders in which an operand is a

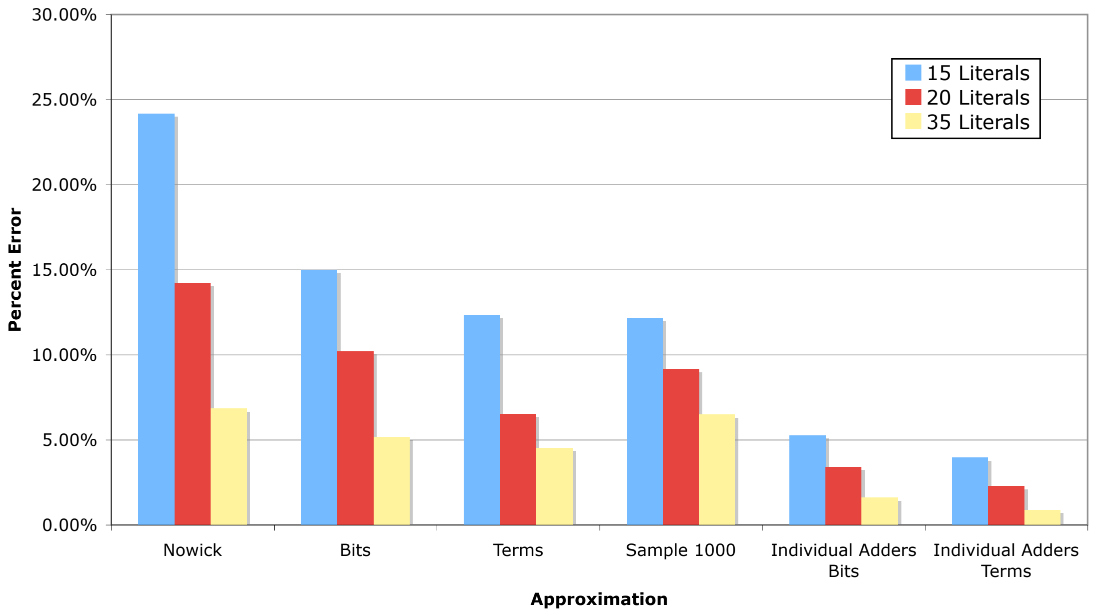

Figure 9: The average percent error for various lateness approximation functions with different constraints on the number of literals. As expected, as the number of literals increases, the error decreases. The use of application specific information has the largest benefit when the size of the function is most constrained. The application-specific spatial computing models outperform the 35 literal uniform function using 20 fewer literals.

ADD ID	Bitwidth	Const	Delay Early	Delay Late	Delay Term Logic
112	32	N	0.56	0.73	0.2
131	32	Y	0.47	0.63	0.1
15	32	Y	0.47	0.63	0.1
173	17	N	0.55	0.68	0.2
387	32	Y	0.39	0.59	0.1
551	18	Y	0.46	0.59	0.1
55	32	Y	0.47	0.63	0.1
Brent-Kung 0.66 ns					

Table 1: The performance of “gsm_d” adders.

constant (see column Const). All but two (173 and 551) are 32-bit adders. The matching delays for the early and late completion for each individual adder are shown in columns *Delay Early* and *Delay Late*, while the delay of each termination-logic network is shown in the last column. For comparison purposes, the delay of the standard Brent-Kung adder is listed in the last table entry.

There are two important conclusions. First, the latency of the termination logic is much less than that of the early delay for the adder, which means that the adders are correct. Second, notice that the delay of each adder variates with the size of the inputs, as well as with one of the inputs being a constant. The

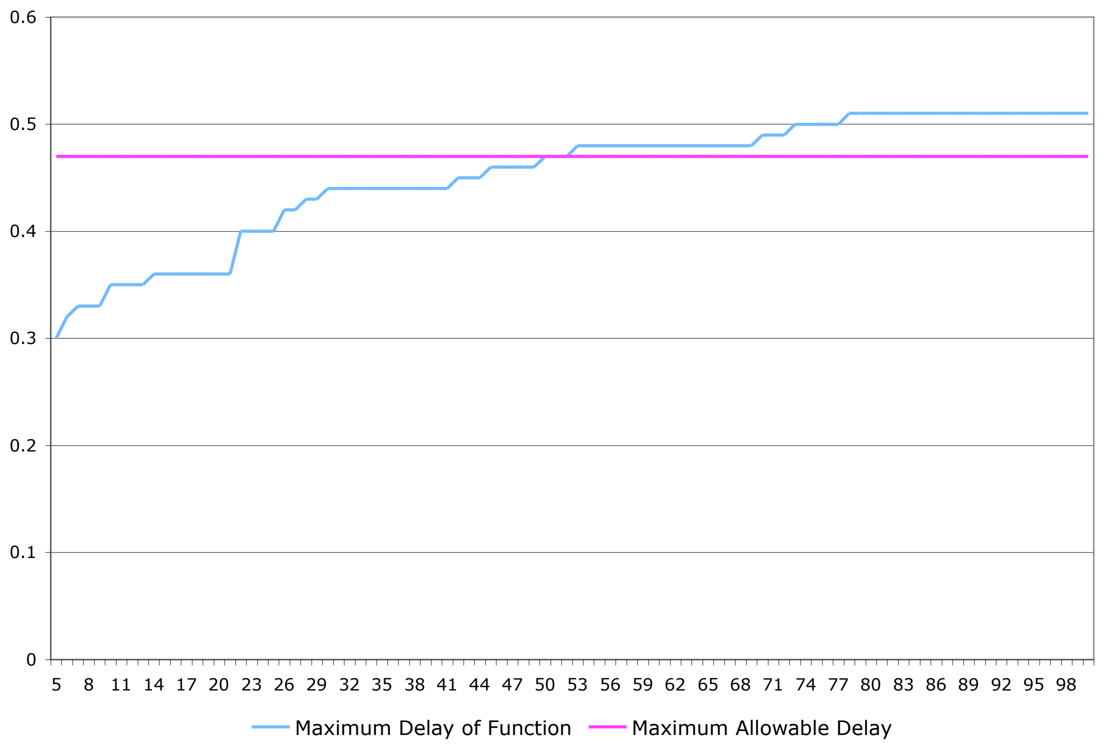

Figure 10: The delays for technology-mapped randomly-generated termination logic functions. Each point is the maximum delay across 20 randomly generated functions. The "maximum allowable delay" is the delay required such that level-3 generate signal can used in sum computation.

adder-generation algorithm performs some very simple optimizations when one of the inputs is a constant; however, these optimizations are limited only to the first level of logic in the adder.

Figure 2 shows the area of each `gsm_d` area, both the standard version and the speculative completion one. The average area overhead is 15%. Notice that, even if two speculative adders have the same bitwidth, the area may be different: it is influenced by the size of the termination-logic function and by whether the adder is a constant adder. The last entry in the table indicates the area for the $L_{3\times5}$ speculative adder in [13]. With all the transistor resources now available on chips, we believe that the area increase is a good tradeoff with speed.

Figure 3 shows the power consumption for each adder in the `gsm_d` benchmark, both the standard Brent-Kung adder and the speculative adder. On average, the speculative adders consume 16% more power than the non-speculative one. In comparison, the power consumption overhead for Nowick's speculative completion adder with the $L_{3\times5}$ is lower (12%).

6 Conclusion

This paper presents a new method for improving the speed of additions by using application-specific information. It builds on "speculative completion" [13], but our experiments show that it results in better early-detection rates than the original design because it uses detailed knowledge of addition data profiles,

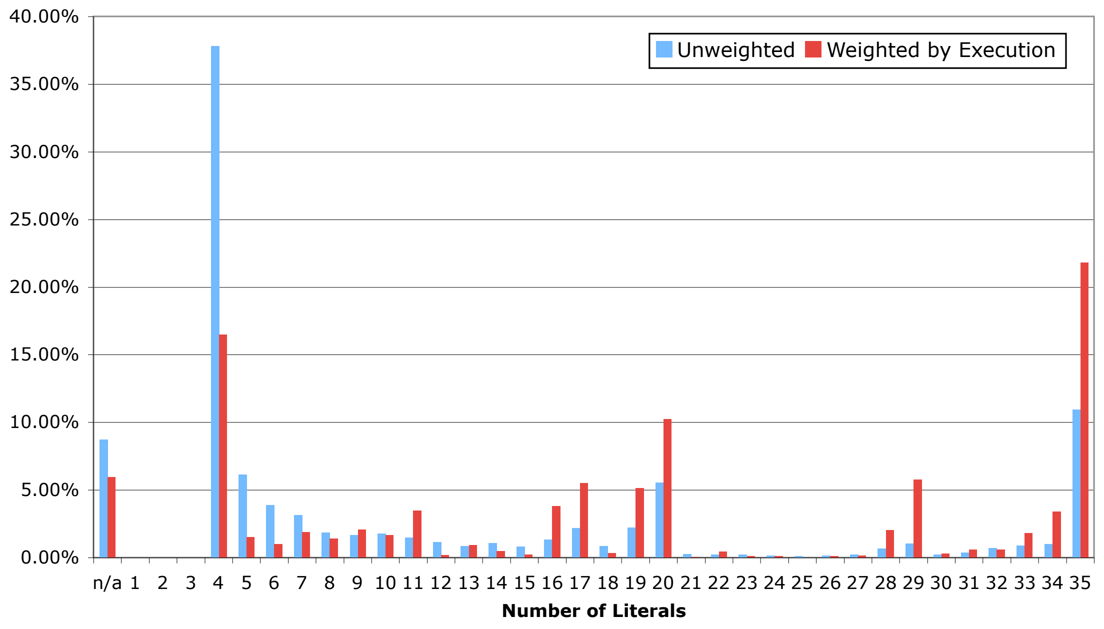

Figure 11: The distribution of the size of lateness approximation functions across all adders of all the benchmarks. The unweighted distribution counts each adder equally whereas the weighted distribution weights each adder by its execution frequency. Adders which do not demonstrate detectable early termination behavior frequently enough to benefit from early termination fall in the n/a category.

ADD ID	Standard Adder (μ^2)	Speculative Adder (μ^2)	Area Overhead
112	112.65	135.17	20.00%
131	91.67	102.73	12.06%
15	91.67	102.24	11.53%
173	56.73	71.56	26.14%
387	91.67	97.16	5.99%
551	48.66	57.26	17.68%
55	91.67	102.73	12.06%
Avg:	95.55	83.53	15.07%
Nowick	112.65	135.94	17.13%

Table 2: Area for `gsm_d` adders: the standard versions and the speculative versions.

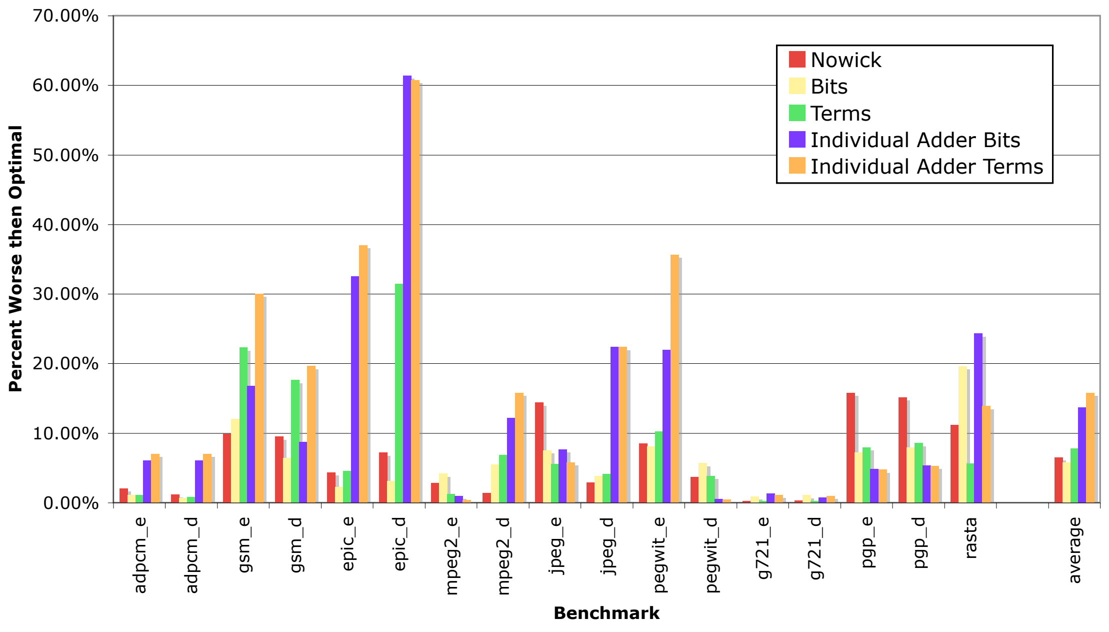

Figure 12: The percent error of various lateness approximation functions when evaluated on program executions using different inputs than the profiled execution. A maximum of 35 literals per a function is allowed. In some cases, the approximation functions remain effective, but in others the error dramatically increases compared to that of Nowick.

ADD ID	Standard Adder (mW)	Speculative Adder (mW)	Power Overhead
112	4.91	5.52	12.42%
131	4.83	5.34	10.56%
15	4.34	5.47	26.04%
173	3.35	4.01	19.70%
387	4.78	5.45	14.02%
551	3.48	4.13	18.68%
55	4.87	5.45	11.91%
Avg:	4.37	5.05	16.19%
Nowick 4-lit	4.91	5.52	12.42%

Table 3: Power consumption for `gsm_d` adders: the standard versions and the speculative versions.

while, at the same time, produces termination logic networks which have fewer literals than the original and are thus faster and smaller. Profiling techniques are powerful, but the data sets must be large and complete enough such that these techniques are effective.

A second contribution of our paper is a modified Brent-Kung adder implementation which is more efficient for speculative completion techniques. This adder can be easily implemented with standard gates, and is amenable to automatic generation by CAD tools.

In summary, this paper demonstrates the power of combining asynchronous circuits and ASH. By using high-level information (e.g. bit-width analysis and profiling information) one can optimize low-level circuits (e.g. early termination networks), resulting in improved overall performance.

References

[1] Richard P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, 1982.

[2] Eric Brunvand. The nsr processor. In *26th HICSS*, 1993.

[3] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[4] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *European Conference on Parallel Processing (EUROPAR)*, volume 1900 of *Lecture Notes in Computer Science*, pages 969–979, Münich, Germany, 2000. Springer Verlag. An expanded version is in TR 00.

[5] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *ASPLOS*, Boston, MA, October 2004.

[6] Steve B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V Woods. A micropipelined arm. In *VLSI 93*, pages 5.4.1 – 5.4.10, September 1993.

[7] Seth Copen Goldstein and Mihai Budiu. NanoFabrics: Spatial computing using molecular electronics. In *International Symposium on Computer Architecture (ISCA)*, pages 178–189, Göteborg, Sweden, 2001.

[8] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[9] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[10] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1:1:119–137, July 1992.

[11] S.M. Nowick. Design of a low-latency asynchronous adder using speculative completion. In *IEEE Proceedings - Computers and Digital Techniques*, volume 143-5, pages 301–307, September 1996.

[12] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, 1993.

[13] Steven M. Nowick, Kenneth Y. Yun, Ayoob E. Dooply, and Peter A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 210. IEEE Computer Society, 1997.

[14] Ad MG. Peeters. *Single-rail handshake circuits*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[15] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *ASYNC'00*, pages 198–209, April 2000.

[16] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[17] Steve H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., 1969.

[18] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic synthesis (IWLS)*, pages 501–508, Temecula, CA, June 2004.

[19] Tom Verhoeff. *A theory of delay-insensitive systems*. PhD thesis, Technische Universiteit Eindhoven, 1994.