# Power-Aware and Branch-Aware Word-Length Optimization

W.G. Osborne, J.G.F. Coutinho, W. Luk, O. Mencer
Department of Computing
Imperial College, London
{wgo, jgfc, wl, o.mencer}@imperial.ac.uk

## Abstract

*Power reduction is becoming more important as circuit size increases. This paper presents a tool called PowerCutter which employs accuracy-guaranteed word-length optimization to reduce power consumption of circuits. We adapt circuit word-lengths at run time to decrease power consumption, with optimizations based on branch statistics. Our tool uses a technique based on Automatic Differentiation to analyze library cores specified as black box functions, which do not include implementation information. We use this technique to analyze benchmarks containing library functions such as square root. Our approach shows that power savings of up to 32% can be achieved on benchmarks which cannot be analyzed by previous approaches, because library cores with an unknown implementation are used.*

## 1 Introduction

This paper focuses on minimizing the power consumption of circuits by adapting the word-lengths of variables at run time, as opposed to design time, based on the branching characteristics of the design.

Using accuracy-guaranteed word-length optimization [15], we are able to produce designs that minimize power consumption while guaranteeing specified output accuracy requirements. We guarantee the error characteristics of the program including the average, best and worst case errors under certain conditions. We then extend this static approach to include run-time power optimization by analyzing the branching behavior of programs.

We start from a C/C++ description of a design. This design may contain single or double-precision arithmetic and branches described using `if` statements, `switch` statements and loops. Loops can be thought of as branches because the execution path will change depending on the loop condition. We assign a set of errors to each branch of the program based on statistical analysis of the program's control-flow, designed to minimize the overall error and area of a circuit design; if a block of code is executed frequently, it is likely to contribute more to the output error.

To reduce the accuracy of code blocks we decrease the precision of arithmetic operators, which has the same effect as reducing the number of bits used for the variables (Section 5.3). To extend this method to work on dynamically changing input data, we gather branch frequency statistics at run time. This enables the circuit to evolve based on input data, resulting in designs that consume less power than those produced using previous accuracy-guaranteed approaches which do not adapt to input data changes.

Several of our benchmarks use complex functions, such as square root. In order to implement these functions efficiently in hardware, specific implementations must be used depending on the device. This makes analyzing their errors difficult, because the implementation may be unknown. To handle this we extend a dynamic word-length optimization technique called Automatic Differentiation to calculate the precision of these operators (Section 5.2).

To assist design exploration, we instrument the code to automatically determine the input ranges. This extension to our previous work means that the process only requires a single metric from the user specifying the accuracy, for example, the output precision. Our approach involves instrumenting code and gathering information about internal ranges, which can be used to determine the representation that would best suit the variables (Section 5.1).

To summarize, the novel elements of our approach include:

1. Dynamic range analysis and control-flow analysis to improve static word-length optimization (Section 4).

2. The application of clock gating to word-length analysis allowing word-lengths to change at run time to produce low power designs (Section 5.4) based on branch statistics and loop iteration bounds, which are not known at compile time (Section 5.3).

3. The use of Automatic Differentiation to analyze the word-lengths of black-box functions (Section 5.2).

4. Demonstration of our approach on 5 benchmarks achieving power improvements of up to 32%: ray-tracing, molecular dynamics simulation, simulation of guitar string motion, Discrete Cosine Transformation (DCT), and B–Splines (Section 6).

## 2 Background and Related Work

### 2.1 Word-Length Analysis

Two common methods of storing numeric values on FPGAs are using the fixed-point and floating-point formats. Floating-point units can be used for a wider range of applications than fixed-point units because they can be used when a large dynamic range is required as well as high accuracy. The disadvantage is that they are larger and slower than fixed-point units. For this reason fixed-point units are preferred in FPGA architectures. Since software programs commonly use floating-point, we must convert between the two formats.

In order to do this we must calculate a range and precision for each variable, which can be done statically or dynamically. Static techniques tend to be more conservative since they do not have access to specific input data sets and may therefore overestimate the range and precision. In contrast dynamic analysis yields more accurate results for the given test data, but is not guaranteed to be accurate for any input data set. The static word-length analysis technique we use in this paper is designed to guarantee the accuracy on the outputs: given an output error requirement, the technique can be used to calculate range and precision values to meet this. An example of how to calculate the error of the function $y = (a \times b) + c$, can be seen below:

$$\begin{aligned} multiply_{error} &= (a_{error} \times b_{error}) + \\ &\quad (a_{error} \times b_{max\ range}) + \\ &\quad (b_{error} \times a_{max\ range}) + \\ &\quad (2^{-precision\ of\ multiply}) \\ y_{error} &= multiply_{error} + c_{error} + \\ &\quad 2^{-precision\ of\ y} \end{aligned}$$

where the errors on the inputs are defined as $2^{-precision\ of\ input}$. For more information about the algorithm, see [15].

### 2.2 Power

Power consumption in FPGAs can be characterized as static or dynamic. In this paper we show reductions for dynamic power. Dynamic power consumption is caused by signal transitions (signal toggling) and can be modelled as:

$$P = \sum_{i \in resources} C_i V_i^2 f_i$$

where $C_i$, $V_i$, and $f_i$ are the capacitance, voltage swing, and operating frequency of resource $i$, respectively [19]. The speed of the circuit and the input data used are important when determining power consumption, because they both affect the signal transition rate.

### 2.3 Function Analysis

We use Automatic Differentiation to work out the word-lengths of black-box functions. The following equation describes how input and output sensitivities are related:

$$\begin{aligned} y &= f(x) \\ \Delta y &= \Delta x \times \frac{dy}{dx} \end{aligned}$$

The output sensitivity is given: this is related to the precision of the output. Using a data-flow graph the errors are propagated from the outputs to the inputs. Once the sensitivities are known for each variable, precisions are assigned to them. For more information see [2].

### 2.4 Related Work

In control-flow analysis, Styles and Luk [17] use information about branch frequencies to reduce hardware resources for implementing branches that are infrequently taken. Since program executions often change their behavior based on input data, the circuit needs to evolve at run time to keep error to a minimum.

Bondalapati and Prasanna [5] reconfigure the circuit at run time and show that it can be used to reduce execution time by up to 37%.

As mentioned in the introduction, we start from a high-level description. Some researchers [4, 11] use high-level models to optimize design area and power consumption. In order to model the components accurately, low-level characteristics have to be captured. Since we may not know in advance which chip the design will be synthesized for, we cannot use these approaches directly. Clarke *et al.* [7] present power models for addition and multiplication which do not depend on their implementation, but instead on the input data supplied. If the characteristics of the input signal are not known in advance, this method cannot be used.

Zhang *et al.* [20] analyze the effect of clock gating on power efficiency showing that FPGAs, although not as efficient as ASICs, can achieve significant power reductions.

Word-length optimization has been used to reduce power consumption on FPGAs and ASICs [9, 14]. Constantinides [8] shows that word-length optimization can reduce power consumption by looking at the sensitivities of variables to small errors. This approach reduces power consumption as a side-effect of reducing area.

Abdul Gaffar *et al.* [1] present an approach to reduce dynamic power by over 10% by using analytical models of power consumption. They show that area-optimized designs will not always be the most power-optimal. Each design is optimized independently and no consideration is given to using a single functional unit for several different word-lengths.

Lee *et al.* [12] have developed a system called MiniBit, which employs static analysis to produce accuracy-guaranteed results. This system does not use information about control-flow to improve the errors in different parts of the program.

In contrast, Abdul Gaffar *et al.* [2] use a dynamic approach to work out the precision of variables. This uses information gathered at run time, but cannot be used to work out the precision of functions that are more complicated than arithmetic operators such as addition and multiplication. Our approach can determine precision of any unknown function. Both approaches will reduce power consumption of a design as a side-effect of reducing area.

## 3 Design-Flow

Our design-flow, shown in Figure 1, has three inputs. The first is a C/C++ design annotated with the error constraints on the output variables. The second is a set of error constraints on variables in the program. While any variable may have a constraint associated with it, all output variables must have a constraint specifying their precision. This enables the tool to determine when to stop reducing the precision of a variable. The third input is either input data or a set of input variable ranges for the program. This information is required to analyze the precision of variables in the system (Section 4.1). If an input data set is specified, the input ranges can be collected automatically using our tool. Hence in this case the developer only needs to supply the input data and output precisions.

The *PowerCutter* tool produces two outputs. The first is a C/C++ design annotated with the range and precision word-lengths of all of the variables. This information is used to produce fixed-point designs. We use fixed-point because the range of the variables in our design is small in general. In this case fixed-point units will be more power-efficient and smaller than floating-point units.

The second output is a database of statistics which can be used in conjunction with the annotated C/C++ design to generate hardware with word-lengths adapting to run-time conditions. This run-time adaptable hardware can be implemented by run-time reconfiguration [5], or by activating only the optimal number of bits required for each variable using a

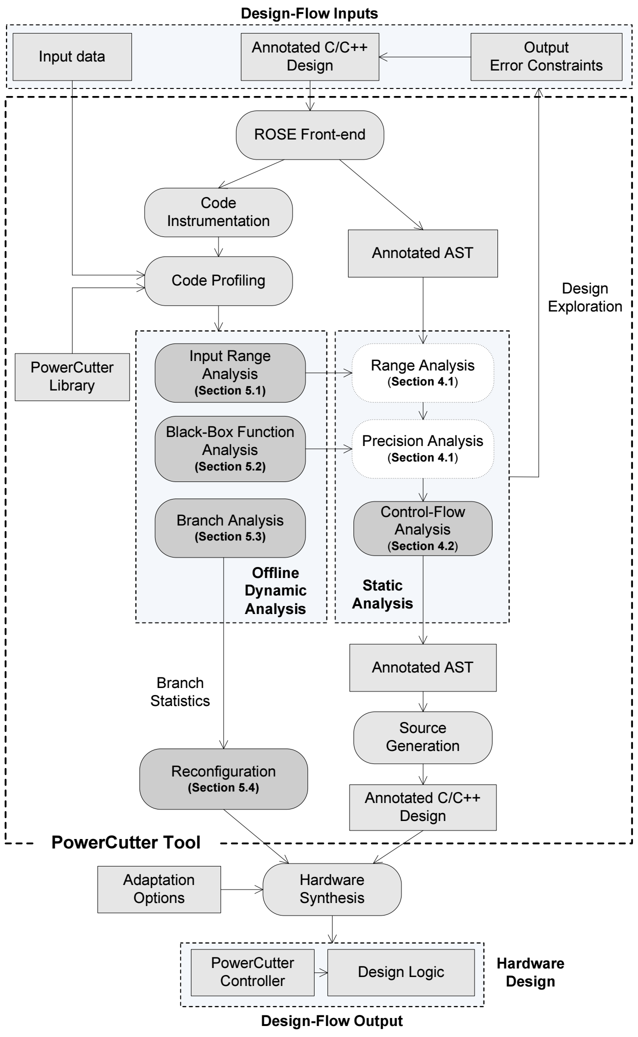


**Figure 1. An overview of the PowerCutter design-flow.**

clock gating technique (Sections 5.3 and 5.4). The former requires a smaller area, but can take longer to reconfigure; the latter does not achieve an area reduction but supports fast word-length adaptation.

The first stage of the design-flow is to parse the source code using the ROSE [16] front-end. The static analysis is the core analysis used by the tool. It starts with range analysis (Section 4.1) to determine the range of every variable being analyzed. These results are passed to precision analysis (Section 4.1) which determines the number of fractional bits required for each variable. Control-flow analysis (Section 4.2) is applied which weights the results based on the branch characteristics.

Offline dynamic analysis provides information to the static analysis. The first stage involves instrumenting the code with calls to the *PowerCutter* library in order to supply information about the code when it is profiled. The code is also instrumented with *truncate* calls to look at the outputs at different precisions so that the developer does not need to supply output precisions to our tool, and only needs to supply input data to the program being analyzed.

Input range analysis (Section 5.1) automates the process of collecting input range information, and can also detect patterns in these ranges. The next stage is black-box function analysis (Section 5.2) which is designed to determine the precision of functions that do not have a specified implementation such as a square root IP core. Branch analysis (Section 5.3) gathers statistics about the frequency of branches and patterns in their execution. This information can be used to generate a hardware description of a controller to predict the direction of branches and gather additional branch statistics while the circuit is running. This is combined with our arithmetic modules which include logic to reduce the precision at run time, to produce the final hardware design.

Design exploration allows the output error requirements or input ranges to be changed. It is also possible to constrain intermediate ranges and precisions, for example to set an upper bound on the number of bits used for a variable.

## 4 Static Analysis

### 4.1 Range and Precision Analysis

Every variable involved in arithmetic operations has a range and a precision associated with it. Our approach to range and precision analysis is accuracy-guaranteed, which means that any operations performed by the static analysis are guaranteed to produce a specified accuracy irrespective of the input data. Since these results will be conservative, dynamic analysis can be used so that the results are guaranteed for a specific input data set. Our range analysis uses a combined Interval/Affine analysis. Starting from the input ranges, operations are performed on ranges as opposed to variables in order to produce ranges for the intermediate and output variables. Affine arithmetic is used because it maintains correlations between the ranges, resulting in smaller output ranges if a variable is used in several places. Section 2.1 describes how errors are used instead of numeric values to calculate the worst-case error on the output, if two input errors are specified to a function, such as multiplication. Range analysis is performed first because variable ranges are required in precision analysis. This range and precision analysis enables us to perform a floating-point to fixed-point conversion if required.

We target high-cost resources first, for example multipliers and dividers and of these, reduce the resources with the lowest error. For more information see [15].

### 4.2 Control-Flow Analysis

When analyzing the size of variables, it is important to assign a cost to a reduction. A multiplication operation will typically have a higher cost than an addition operation because it takes up more area and is slower. We combine traditional approaches to word-length analysis [12, 15] with control-flow analysis. If one block of code is executed more often than another block, its cost should be higher because it will take up more execution time and is likely to contribute more to the overall error.

We weight the cost analysis based on the proportion of time that a block of code takes to execute. This on its own will only give a small performance improvement, but we combine it with dynamic analyses to reduce the power consumption of the design (Section 5.3). Since we reduce the precision of blocks of code that contribute more heavily to the execution time and we reduce the precision at run time (when the circuit is running as opposed to our offline analysis), there will be a greater power saving than if we reduce the precision of blocks that only contribute a small proportion of the execution time. This optimization therefore has a small effect on its own but a large effect when combined with dynamic analysis discussed in Section 5.3.

We also have the option of weighting the error function. Figure 2 shows a situation in which it may be more cost effective to weight the error function. If the if-condition is only executed 50% of the time, the error on *accumulate_a* will be twice what it should be.

```
if (condition) {
  accumulate_a *= a;
} else {
  accumulate_b *= b;
}
```

**Figure 2. Weighting the error function to handle control-flow.**

## 5 Dynamic Analysis

Static analysis on its own produces conservative results. To combat this we use dynamic analysis which extends our previous work, in which only loops are analyzed [15]. We add three new optimizations, performed in the following order:

1. Dynamic range optimization of variables (Section 5.1).
2. Analysis of black-box functions to calculate precision without knowing their implementation (Section 5.2).

3. Use of control-flow information gathered at run time to minimize power consumption (Section 5.3).

## 5.1 Input Range Analysis

We start by instrumenting every assignment statement in the code with a call to the *PowerCutter* library, unless the expression corresponds to an address. When the code is executed, the library keeps track of all of the variables and their corresponding range. The impact of this is that our design-flow only requires a quality metric for the output, which may simply be an output precision. If all of the ranges have not been gathered, the ranges are filled-in using a combined Interval/Affine range analysis [15].

In some cases, it may not be cost effective to use a fixed-point number system. To handle this, our library also keeps track of the range at different points. When a range is input into the library, the $log$ is taken. This means that when a histogram of the data is plotted, the number of points around zero is greater, since these are the most important values. The values around zero enable us to determine whether floating-point will be more efficient or whether the values can be shifted. Very small values may not be able to be implemented in fixed-point efficiently because there are a large number of bits that will always be zero. To avoid this, either floating-point or a shift to remove the zeros can be used. Our system can also be used to detect patterns in input ranges which may be used to save power if the range can be predicted.

## 5.2 Black-Box Function Analysis

Our previous paper shows how information gathered at run time can be used to make the results less conservative. We expand this technique to calculate the precisions of functions that are unknown. In our designs we use square root and divider modules which cannot have their error calculated, because we don't know how the function is implemented in hardware. We use a technique called Automatic Differentiation which has been used to analyze word-lengths [2]. We extend this method to differentiate any function, beyond standard operators, for example add and multiply.

To demonstrate this approach, consider the expression $y = a \times b$.

$$\begin{aligned}\frac{dy}{da} &= \frac{(a \times b) - (a_{old} \times b)}{a - a_{old}} \\ \frac{dy}{db} &= \frac{(a \times b) - (a \times b_{old})}{b - b_{old}}\end{aligned}$$

where $a_{old}$ and $b_{old}$ are the previous values of $a$ and $b$ respectively.

Using these derivatives, we can calculate the sensitivity of the inputs, given the output sensitivities:

$$\begin{aligned}a_{error} &= \frac{y_{error}}{2 \times \frac{dy}{da}} \\ b_{error} &= \frac{y_{error}}{2 \times \frac{dy}{db}}\end{aligned}$$

The derivative is multiplied by 2 because we divide the error equally between $a$ and $b$ in this case. Given the errors on each input, we can obtain the word-lengths. To perform the calculations illustrated above, we instrument the code being analyzed with calls to the *PowerCutter* library.

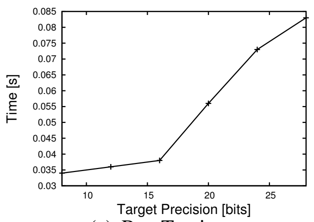
An alternative to this approach is to use function approximation [13]. This involves designing functional units that can be analyzed statically to calculate accuracy-guaranteed designs. The disadvantage with this is that the functional units may not be as efficient as optimized cores. We show that this technique can be applied in Section 6.3 to analyze a square root.

## 5.3 Branch Analysis

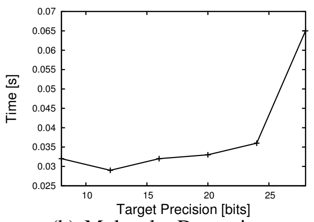
To optimize branches we first look at the control-flow. We break the program up into a set of basic blocks; each block having one entry point and one exit point. If a block of code is executed a large number of times, we assume that it will have a greater influence on the error of the final result. If a block is not called many times it is likely that the error contributed by that block will be small, so we may be able to reduce the precision of the values used in this block.

Each branch is assigned an error metric based on its frequency; the higher the frequency, the lower the error metric. One further approach is based on branch prediction. If a branch is very unlikely to be taken, the precisions of variables along that branch could be minimized to reduce the area of the circuit. This can be extended to handle loops as well. In our last paper [15] we discuss loop analysis as a method to enhance the static word-length analysis. It works by reducing the precision required for loops by estimating the number of loop iterations. This is an offline dynamic analysis that uses ROSE [16] to instrument the source code. If the number of iterations is not known at compile time, we reduce the word-lengths at run time to save power for the design. This also has applications to custom processors in which the word-lengths may need to change based on the program running on them. Our tools can be employed to dynamically change the word-lengths at run time based on user-supplied stimuli.

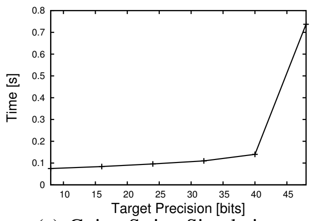
Convergence is another application that can benefit from reduced precision arithmetic. The more iterations of the loop that execute, the more precise the calculation needs to be. The ability to reduce the precision of the arithmetic operators can therefore be used to reduce the power of such systems. One example of this is the Newton–Raphson method which



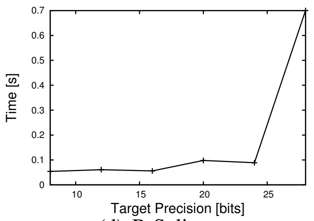
(a) Ray Tracing



(b) Molecular Dynamics

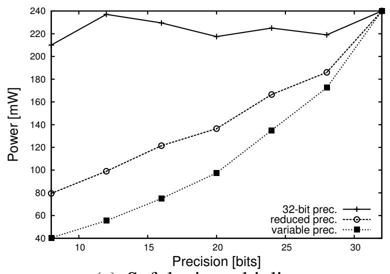


(c) Guitar String Simulation

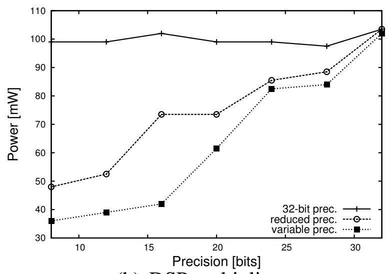


(d) B-Splines

**Figure 3. Time before full run-time reconfiguration becomes more efficient than reconfiguration strategies in which the entire design resides on the chip.**



(a) Soft logic multiplier



(b) DSP multiplier

**Figure 4. Power saving for a 32-bit multiplier by reducing the precision.**

calculates an approximation for the roots of functions. This is quadratic which means that the number of bits doubles at every iteration.

## 5.4 Reconfiguration

In order to save power and to allow a design to adapt to different input conditions, we explore 3 methods of reconfiguring the design. This means that the design can adapt to changing conditions, enabling it to have a high or low accuracy depending on the output accuracy requirement. When a lower accuracy is required we can reconfigure the design to save power. Using a lower accuracy can also allow multiple inputs and results to be combined to reduce bandwidth, accelerating the application.

### 5.4.1 Clock Gating

Components for clock gating, such as BUFGCE on Xilinx devices, are few in number on Virtex FPGAs (16 on a Xilinx XC2VP30 FPGA), so we use a different method to gate the clock. BUFGCE is a global clock buffer with a single input and clock enable. To enable more local clock gating, which is required because we are only gating part of an arithmetic operator, we can use the clock enable on the flip-flops. We find a 45% drop in power consumption when lowering the word-length from 32 bits to 16 bits with an 11% increase in the number of slices. Because we only use the clock enables on flip-flops to save power, the clock is still toggling and therefore consumes more power than an ASIC design in which the clock can be completely gated.

Another option is to use a more global clock gating technique whereby we select entire arithmetic operators, for example multipliers with different precisions, instead of gating individual bits; we do not do this here because it requires a large area. An even more global technique is to use run-time reconfiguration. This means that the whole FPGA, or just part of it, is reconfigured at run time. With a smaller design on the chip, the power consumed will be lower. The disadvantage is that there are speed and power overheads associated with the reconfiguration process.

### 5.4.2 Run-Time Reconfiguration

Given that $energy = power \times time$, we can look at the amount of time required before run-time reconfiguration will reduce the overall energy of the system.

Figure 3 shows how long it takes before the overhead of reconfiguring the design by keeping the entire design on the chip, becomes more costly than reconfiguring the entire circuit. When applying our model, we use 14ms [10] for the average reconfiguration time and 1500mW [3] for the average reconfiguration power; the energy required to reconfigure the chip is therefore 21mJ. The overheads of using clock gating are taken from the designs in Section 6. If the reconfiguration interval required is of the same order of magnitude as the clock cycle time, a clock gating approach may be more suitable.

### 5.4.3 Changing Signal Transition Rates

The reconfiguration method found to be the most efficient involves reducing the signal transitions by feeding zeros into the unwanted bits. Figure 4 shows the power consumption of a 32-bit multiplier running at 200MHz with different precisions, changed by setting the unwanted bits to zero. This method has a 2% area overhead because the inputs must be multiplexed.

## 5.5 Annotations

To use this system, we add annotations to the code being executed on a custom processor or transformed to run on a different architecture for example an FPGA.

```
#pragma output_precision acc_a:24
float acc_a = 1;

for (int i = 0; i < 10; i++) {
  if (condition) {
#pragma executed 0.5
    acc_a *= a;
  }
}
```

**Figure 5. Annotation example.**

The `pragma` annotations above tell our tool that the output precision should be 24-bits (guaranteed) and that the if-condition is executed 50% of the time. Alternatively this can be calculated automatically by using the offline dynamic analysis described in Section 5.1.

## 6 Case Studies

## 6.1 Experimental Setup

To perform the experiments we use the Xilinx XUP board with a Virtex II Pro XC2VP30–7 FPGA. All designs were synthesized using Handel–C 4.1 and Xilinx ISE 9.2. We measure the power consumption by attaching an ammeter to the 1.5V VCCINT jumpers, which only supply power to the FPGA.

## 6.2 B–Splines and 8×8 DCT

Figure 6 shows two power consumption readings for the B–Splines benchmark. The *32-bit prec* line shows the power consumption of the design at varying output precision. The dashed line is the power consumption of the design at varying precision assuming that the entire design is on the chip.

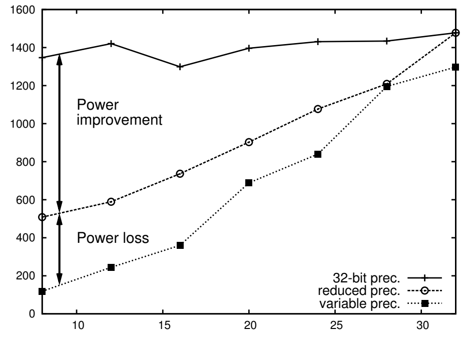


**Figure 6. Power consumption of the B-Splines benchmark.**

Since the entire design is on the chip for *reduced prec* as opposed to the minimum logic required (*variable prec*), there will be a small amount of power loss so there is a gap between the two lines. *Power loss* indicates this and should therefore be minimized. Any Block RAM used in the designs will not be gated because all values must be stored at their maximum precision, and therefore contribute to a smaller power improvement. The ideal case occurs when all of the logic is used in fixed-point calculations and there is no power loss. This would mean that the two lines on the graph would lie on top of each other. Since this will never occur in practice, there is always a small gap.

Because we don't want to include the overhead of I/O in the designs, we simulate the inputs using 8-bit counters to create a uniformly toggling input. Although the toggle rates may be reduced in practice, this will not significantly alter the proportional reduction we get from reducing the word-length.

For the B–Splines design we show power savings of 2.5% per bit of precision (*Power improvement* in Figure 6). Because the signal transition rates have been reduced by setting part of the input to a constant value, both precisions can be used in the design. In each of these designs, the output precision is guaranteed because we use accuracy-guaranteed word-length optimization [15]. We also show power improvements of 2% per bit can be achieved using our approach on the DCT design, compared to previous accuracy-guaranteed approaches [12]. This is lower than the B–Splines design due to some integer arithmetic operations whose precision cannot be reduced.

## 6.3 Ray Tracing

The bottleneck in most ray-tracers is the ray-sphere intersection; for every ray, it must be determined whether the ray will intersect with an object. This kernel is executed 70 million times for each image, of which approximately 2 million calls are intersections. We traverse the rays in a

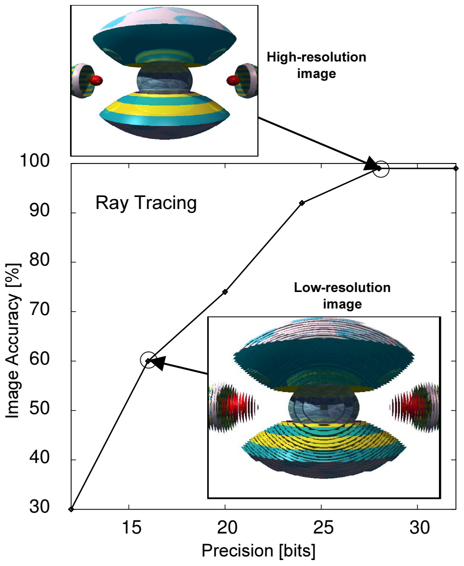
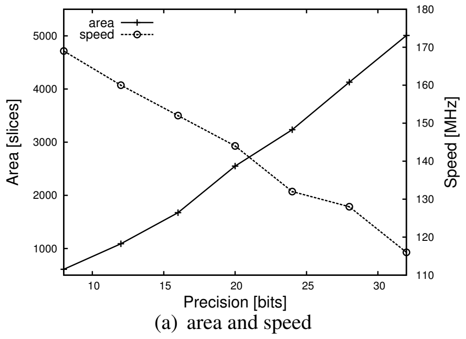
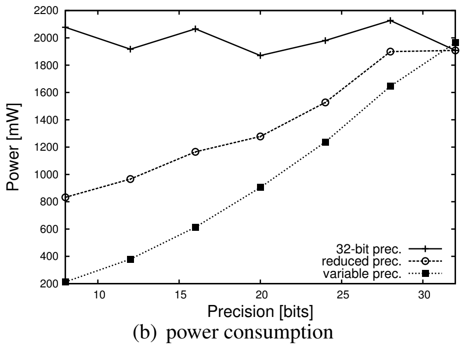


**Figure 7. A comparison of image quality with output precision.**



(a) area and speed



(b) power consumption

**Figure 8. Area, speed and power consumption of the ray tracer at different output precisions.**

breadth-first fashion because this makes the ray intersections more predictable.

The errors in the resulting image, caused by the ray tracer, are most noticeable on the boundaries, giving rise to jagged edges. We can therefore make the assumption that if two consecutive rays intersect, the ray is not on a boundary, and further reduction can occur.

If the hardware design were implemented specifically to cater for a simpler image, the precision could be greatly reduced to conserve power. Since any image can be input to the ray tracer we reduce the word-lengths at run time by gating the clock to reduce power consumption. Based on our static analysis, we choose a set of word-lengths meeting the different error requirements.

Figure 8(a) shows the area and speed of the ray tracer at different output precisions. In general area increases as output precision increases. As the chip starts to fill up, however, the area will stop increasing because the mapper tries to pack the design onto the chip. Figure 8(b) shows the power consumption at different precisions as well as power consumption with logic to reduce the signal transitions included. As explained above, this differs slightly because the logic to implement the design at full precision must be on the chip even when a lower precision is being used. Since this design only has a small amount of power loss, the lines are close together.

This design contains a square root operation which cannot have its precision calculated using standard approaches (Section 5.2). We show power improvements of 2.3% per bit can be achieved using our approach on this application, compared to previous accuracy-guaranteed approaches [12]. This amounts to a maximum of a 32% power saving when the precision of the ray tracer is reduced from 32 bits to 20 bits, the precision at which errors start to become noticeable.

## 6.4 Molecular Dynamics Simulation

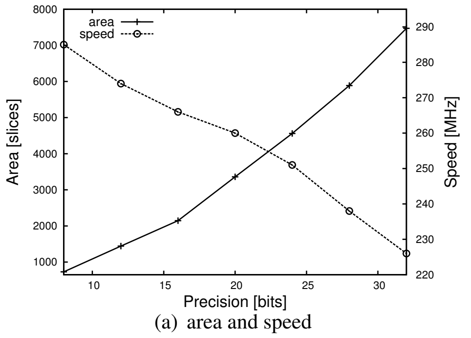
Figure 9 shows an outline of the particle interaction kernel similar to that used in the MolDyn benchmark [6], a molecular dynamics simulation. This example could easily be applied to collision detection since the principle is the same; it contains several conditional statements which may not be taken very often, making it a good candidate for run-time branch prediction to reduce error.

Figure 10 shows how area, speed and power consumption are affected by output precision. We achieve power savings of up to 2.8% per bit. The fluctuation in the curve is due to the use of an area model to calculate the word-lengths as opposed to a power model which may increase area.

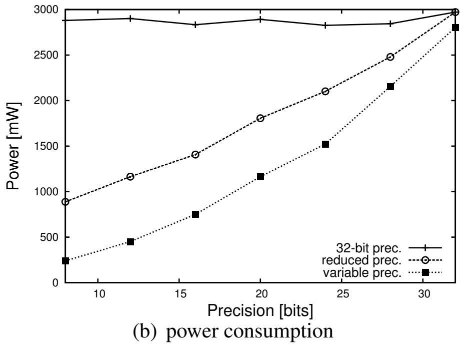
This design highlights an important aspect of our precision analysis algorithm which uses heuristics for word-length analysis. These designs take between 25 and 40 seconds to analyze due to the large number of variables compared to previous accuracy-guaranteed approaches which take over 100 seconds to analyze simpler benchmarks such as the DCT.

```
for (ii = 0; ii < n_inter; ii++) {
  // Distance between particles
  i = intr[ii].first;
  dx = x(i) - x(j);
    ...

  // Check for collisions (6 if branches)
  if (dx < -side) ...
    ...

  dist_sqr = ((dx * dx) + ...;

  // Check for an interaction
  if (dist_sqr < interact_dist_sqr) {
    // Calculate forces
    rdist_sqr = 1.0 / dist_sqr;
      ...
    rdist_sqr7 = rdist_sqr6 * rdist_sqr;
    force = rdist_sqr7 - (0.5 * rdist_sqr4);

    forcex = dx * force;
    fx(i) += forcex;
    fx(j) -= forcex;
      ...
  }
}
```

**Figure 9. Particle interaction.**



(a) area and speed



(b) power consumption

**Figure 10. Power consumption of the molecular dynamics simulation.**

## 6.5 Guitar String Simulation

This kernel simulates a real vibrating string by using the finite difference method.

Figure 11 shows how power consumption changes as a result of changing output precision. The area increases since more logic is required to meet a higher error requirement, which also causes the power consumption to increase. The *variable prec* line in Figure 11 shows how power consumption varies as precision increases. This is a small design, so a large proportion of logic is dedicated to non-arithmetic operations, therefore the effect of dynamic optimization is not as noticeable, although it does yield power savings if a lower output precision is adopted.

We show power improvements of 0.5% per bit can be achieved using our approach. Although this is a lot lower than the previous designs, in part due to the size of the design relative to the overheads, this application converges to a solution, which means that at the start of the computation the word-length can be reduced by a large amount; as the computation progresses, the precision will increase.

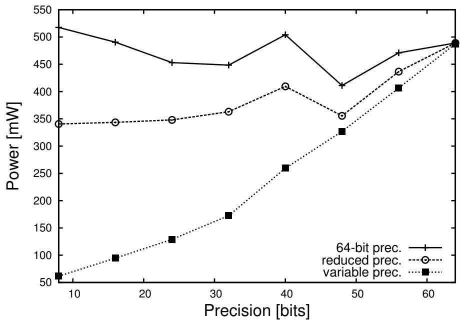


**Figure 11. Power consumption of the guitar string simulation.**

## 7 Conclusion and Future Work

This paper describes an approach to enable word-lengths to dynamically change either by using clock gating or by changing part of the input to a constant value, based on branch probability analysis. The approach achieves up to 32% power reduction for a ray tracing design before a noticeable drop in accuracy can be seen. We gather information about each branch in a program in order to determine the word-lengths to assign to each basic block. Our system can analyze library functions with no implementation specified, enabling us to specify a precision for cores used in our design, with an unknown implementation. This means that we can support library-based designs which previous approaches cannot.

We use input range analysis to detect patterns in the way numbers are used. These patterns are regular over long periods of time; in the ray tracer this may amount to a frame or several frames. Current and future work includes analyzing these patterns further, with the goal of reducing power consumption. We will also explore the trade-offs of different run-time word-length analysis methods, and integrate the proposed approach with other reconfigurable optimization techniques such as program phase analysis [18].

**Acknowledgments.** The support of FP6 hArtes (Holistic Approach to Reconfigurable Real Time Embedded Systems) project, UK EPSRC, the HiPEAC Reconfigurable Computing Cluster, Agility, Celoxica, and Xilinx is gratefully acknowledged. We thank Tim Todman for providing code for the ray tracer.

## References

[1] A. Abdul Gaffar, J. A. Clarke, and G. A. Constantinides. Powerbit - power aware arithmetic bit-width optimization. In *Proceedings of the International Conference. on Field-Programmable Technology*, pages 289–292, December 2006.

[2] A. Abdul Gaffar, O. Mencer, W. Luk, P. Y. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 158–165, December 2002.

[3] J. Becker, M. Hübner, and M. Ullmann. Power estimation and power measurement of Xilinx Virtex FPGAs: Trade-offs and limitations. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*, pages 283–288, September 2003.

[4] S. Bobba, I. N. Hajj, and N. R. Shanbhag. Analytical expressions for power dissipation of macro-blocks in DSP architectures. In *Proceedings of the International Conference on VLSI Design*, pages 358–365, 1999.

[5] K. Bondalapati and V. K. Prasanna. Dynamic precision management for loop computations on reconfigurable architectures. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–258, 1999.

[6] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization and dynamics calculations. *Computational Chemistry*, 4(2):187–217, 1983.

[7] J. A. Clarke, A. Abdul Gaffar, G. A. Constantinides, and P. Y. K. Cheung. Fast word-level power models for synthesis of FPGA-based arithmetic. In *Proceedings of the IEEE Symposium on Circuits and Systems*, pages 1299–1302, May 2006.

[8] G. A. Constantinides. Word-length optimization for differentiable nonlinear systems. *ACM Transactions on Design Automation of Electronic Systems*, 11(1):26–43, January 2006.

[9] F. Fang, T. Chen, and R. A. Rutenbar. Floating-point bit-width optimization for low-power signal processing applications. In *Proceedings of the International Conference on Acoustic, Speech and Signal Processing*, volume 3, pages 3208–3211, May 2002.

[10] B. Griese, E. Vonnahme, M. Porrmann, and U. Rückert. Hardware support for dynamic reconfiguration in reconfigurable SoC architectures. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 842–846, August 2004.

[11] T. Jiang, X. Tang, and P. Banerjee. Macro-models for high level area and power estimation on FPGAs. In *Proceedings of the Great Lakes Symposium on VLSI*, pages 162–165, 2004.

[12] D. Lee, A. Abdul Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1990–2000, October 2006.

[13] D. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54:1520–1531, 2005.

[14] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou. Smart bit-width allocation for low power optimization in a SystemC based ASIC design environment. In *Proceedings of the International Conference on Design, Automation and Test in Europe*, pages 618–623, April 2006.

[15] W. G. Osborne, J. G. F. Coutinho, W. Luk, and O. Mencer. Instrumented multi-stage word-length optimization. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 89–96, December 2007.

[16] D. J. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen. Classification and utilization of abstractions for optimization. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, pages 57–73, 2004.

[17] H. Styles and W. Luk. Exploiting program branch probabilities in hardware compilation. *IEEE Transactions on Computers*, 53(11):1408–1419, 2004.

[18] H. Styles and W. Luk. Compilation and management of phase-optimized reconfigurable systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 311–316, 2005.

[19] G. Yeap. *Practical Low Power Digital VLSI Design*. Kluwer Academic Publishers, 1998.

[20] Y. Zhang, J. Roivainen, and A. Mämmelä. Clock-gating in FPGAs: A novel and comparative evaluation. In *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, pages 584–590, 2006.