Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-042 September 3, 2009

Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures

Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures

Henry Hoffmann Sasa Misailovic Stelios Sidiroglou Anant Agarwal
Martin Rinard

Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{hank,misailo,stelios,agarwal,rinard}@csail.mit.edu

ABSTRACT

Many modern computations (such as video and audio encoders, Monte Carlo simulations, and machine learning algorithms) are designed to trade off accuracy in return for increased performance. To date, such computations typically use ad-hoc, domain-specific techniques developed specifically for the computation at hand.

We present a new general technique, *code perforation*, for automatically augmenting existing computations with the capability of trading off accuracy in return for performance. In contrast to existing approaches, which typically require the manual development of new algorithms, our implemented SpeedPress compiler can automatically apply code perforation to existing computations with no developer intervention whatsoever. The result is a transformed computation that can respond almost immediately to a range of increased performance demands while keeping any resulting output distortion within acceptable user-defined bounds.

We have used SpeedPress to automatically apply code perforation to applications from the PARSEC benchmark suite. The results show that the transformed applications can run as much as two to three times faster than the original applications while distorting the output by less than 10%. Because the transformed applications can operate successfully at many points in the performance/accuracy tradeoff space, they can (dynamically and on demand) navigate the tradeoff space to either maximize performance subject to a given accuracy constraint, or maximize accuracy subject to a given performance constraint. We also demonstrate the SpeedGuard runtime system which uses code perforation to enable applications to automatically adapt to challenging execution environments such as multicore machines that suffer core failures or machines that dynamically adjust the clock speed to reduce power consumption or to protect the machine from overheating.

1. INTRODUCTION

The need to meet hard performance goals has motivated research into systems that trade accuracy for performance or other benefits (such as robustness, energy savings, etc.,). The standard approach to building such applications is manual, ad hoc, and application specific; typically requiring both domain and implementation expertise. For example, MP3 audio encoding uses a lossy compression algorithm designed to reduce the size of data needed to represent an audio recording. The compression works by reducing the accuracy in parts of the sound file that are beyond the range that most people can distinguish. This optimization uses a heuristic that cannot be directly applied to other types of compression (*e.g.* image and video) and does nothing to inform performance/accuracy tradeoffs in other domains like financial analysis. Moreover, the use of standard techniques to optimize a given MP3 encoder implementation requires (in addition to the requisite domain expertise) significant technical skills and familiarity with the code base.

1.1 Code Perforation

We present a novel technique, *code perforation*, that automatically enhances applications to support the management of performance/accuracy tradeoffs. Given a user-specified distortion bound, our implemented SpeedPress compiler automatically identifies parts of the computation that can be discarded (skipped) without violating this bound. For example, if a user is willing to tolerate a .5 dB degradation in signal-to-noise ratio (SNR) for a video, code perforation can automatically identify parts of the video encoder computation that it can skip while decreasing SNR by no more than .5 dB. The result is a computation that performs less work (and therefore consumes fewer computational resources) while still producing acceptable output.

In addition to enabling accuracy/performance tradeoffs, code perforation can enhance robustness and enable energy savings. Our implemented SpeedGuard system adjusts the amount of code perforation in response to changes in the environment (such as core failures or clock frequency scaling) to enable the application to continue to meet its performance goals (in exchange for some distortion in the output) despite the degradation of the underlying computing platform.

1.2 Implementation

SpeedPress performs code perforation by discarding loop iterations, a process referred to as *loop perforation*. The perforating compiler accepts standard C and C++ code and a user-provided method for calculating the output distortion. The compiler then uses profiling to explore the performance/accuracy tradeoff space generated by discarding loop iterations from the original program. Performance improvements are calculated as the speedup (the execution time of the original application divided by the execution time of the perforated application); accuracy changes are calculated using the distortion model presented in [22]. The results of this exploration make it possible to use code perforation to automatically maximize performance subject to a given distortion constraint or to automatically minimize distortion subject to a given performance constraint.

The SpeedGuard runtime system automatically monitors and maintains the performance of applications compiled with SpeedPress.

SpeedGuard uses the Application Heartbeats framework [16] to dynamically monitor the performance of the application as it attempts to satisfy its real-time performance goals. If SpeedGuard detects a drop in performance below the desired threshold, it dynamically increases the amount of perforation to increase performance while minimizing accuracy loss. If performance later recovers, the runtime dynamically adapts to reduce the amount of perforation, potentially switching the application all the way back the original version (which executes with no perforation). SpeedGuard does not rely on detecting specific faults. It instead detects changes in application performance and adjusts the amount of perforation required to accommodate the performance changes. It can therefore adapt automatically to any type of failure that can result in performance loss, making SpeedGuard applicable to a variety of situations such as core failures, frequency scaling, or simply increased load. We envision that this method will be of particular use in real time systems where producing a timely result (even with reduced accuracy) is preferable to violating the application's timing constraints.

We have evaluated our implemented system on a range of applications from the PARSEC benchmark suite [8]. Our experimental results show that code perforation delivers significant performance increases for six of the seven selected applications. Specifically, our performance measurements show that code perforation is able to deliver perforated applications that run between two to three times faster than the original application while producing outputs that differ by less than 10% from the output of the original application. Furthermore, our results show that SpeedGuard can dynamically adapt the amount of code perforation to successfully meet timing constraints in the presence of failures that degrade the underlying computing platform. Specifically, our experimental results show that code perforation makes it possible to meet performance goals in the face of core failures during the execution of a multithreaded application running on an multicore machine. Our results also show that code perforation makes it possible to recover from dynamic clock frequency changes. Code perforation can therefore enable applications to respond productively to overheating (*e.g.* due to fan failure) or changing energy constraints (*e.g.* low battery).

1.3 Scope

There are many applications, for example compilers and some database systems, that have hard logical correctness requirements. We acknowledge that code perforation is not appropriate for these applications because it may cause the application to unacceptably produce incorrect results.

There are also, however, a wide range of applications that can tolerate bounded output distortion. Examples of such applications include applications that process sensory data such as video, audio, and images — for these kinds of applications, the code perforation distortion can either be imperceptible or preferable to the sensory effects (such as jitter or interruptions in smooth content flow) that failure to meet performance goals would induce in the absence of code perforation. Other examples include applications that perform Monte Carlo simulations, information retrieval and machine learning applications, and the wide variety of scientific and economics computations for which the important consideration is producing an output within an acceptable precision range. Code perforation can often acceptably improve the performance of all of these kinds of applications while preserving acceptable output precision. The prominence of these kinds of applications in the PARSEC benchmark suite (which was chosen to be representative of modern performance-intensive workloads) bears witness to their importance in modern computing environments.

1.4 Contributions

This paper makes the following contributions:

- **Basic Concept**: It introduces the concept of code perforation, a general technique that can be automatically applied to enhance applications to support accuracy/performance tradeoffs by selectively discarding computations. An exploration of the resulting accuracy/performance tradeoff space makes it possible to either maximize performance subject to a given accuracy bound or maximize accuracy subject to a given performance bound.
- **Fault Tolerance**: It shows how to tolerate faults in the underlying computing environment by:
  - **Runtime performance degradation detection**: using the Heartbeat API to detect general program performance degradation.
  - **Runtime performance adjustments**: using code perforation to maintain application performance goals by trading off accuracy for increased performance in response to events such as core failures, dynamic changes in environment settings, and increased load.
- **Implementation**: It presents the implementation of code perforation using:
  - **SpeedPress:** A LLVM-based compiler which exploits code perforation to trade accuracy for performance.
  - **SpeedGuard:** A runtime system which dynamically enables code perforation to provide fault tolerance.
- **Evaluation**: It presents experimental results from applying code perforation to several benchmark applications from the PARSEC suite. It also presents the experimental evaluation of code perforation as a fault tolerance mechanism.

The remainder of this paper is organized as follows. Section 2 presents a simple example which illustrates the key concepts of code perforation. Section 3 discusses the implementation of SpeedPress, a perforating compiler for C and C++ programs. Section 4 presents the SpeedGuard runtime. Section 5 discusses our evaluation methodology. Section 6 presents the results of applying the perforating compiler to several PARSEC benchmarks. Section 7 presents the results of the fault tolerance experiments. Section 8 discusses related work. Finally, the paper concludes in Section 9.

2. EXAMPLE

We next present an example that illustrates the use of code perforation to increase the performance of the open-source x264 implementation [32] of the H.264 video encoding standard. Video encoders take a stream of input frames and compress them for efficient storage or transmission. The quality of a video encoder is typically measured using the peak signal-to-noise ratio (PSNR) and bitrate (or size) of the encoded video.

One of the keys to achieving good video compression is exploiting similarities between consecutive frames. The process of finding these similarities is called motion estimation. During motion estimation the frame currently being encoded is broken into 16×16 regions of pixels called macroblocks. For each macroblock, the encoder attempts to find a similar 16×16 region in a previously encoded reference frame. H.264 allows macroblocks to be further broken down into sub-blocks, and motion estimation can be performed on sub-blocks independently. x264 calculates the similarity for macroblocks and sub-blocks by computing the sum of Hadamard transformed differences (SATD) between the

```
static int pixel_satd_wxh(pixel_t *current,
                          int cur_stride,
                          pixel_t *reference,
                          int ref_stride,
                          int w,
                          int h)
{
  int value = 0;
  int i, j;

  short temp[4][4];

  for( i = 0; i < h; i+=4 ) {
    for(j = 0; j < w; j+=4 ) {

      // Performs element-wise subtraction of the
      //  reference frame and the current frame
      element_wise_subtract(temp, current[j], cur_stride,
                              reference[j], ref_stride,
                        4);

      // Performs an in-place Hadamard transform on the
      // difference computed in the previous step
      hadamard_transform(temp, 4);

      // Sum the absolute values of the coefficients
      // of the Hadamard transform
      value += sum_abs_matrix(temp, 4);
    }
    current += 4*cur_stride;
    reference += 4*ref_stride;
  }

  return value;
}
```

Figure 1: Code to compute sum of Hadamard transformed differences. This function is important in video encoding and a good candidate for code perforation.

macroblock (or sub-block) and candidate regions of the reference frame. Figure 1 presents a C function for computing the SATD between two regions.

To find the best match for a macroblock, an encoder would have to search the entire reference frame, but searching such a large area is prohibitively expensive in practice. Even searching every location in a relatively small region of the reference frame can impose an unacceptable performance burden [14]. In practice, motion estimation algorithms typically use clever heuristics to move from one search location to another without having to examine every possible location.

Development of these algorithms is an active area of research in video processing. The common evaluation metric of these algorithms is the reduction in video quality over the raw input and the amount of additional bits required to encode the video. This inherent tradeoff between quality and performance makes motion estimation a potentially good candidate for code perforation.

To evaluate the effect of perforation on accuracy, the compiler needs a standard method for determining an *acceptability model*. This model has two parts; the first is an *output abstraction* or a method for mapping the output of the program to a numerical value or values. In some cases these values are selected directly from the output, in others these values are computed from output values. The second component is *distortion*, which is a measure of how much the output abstraction of the perforated code differs from that produced by the unperforated version.

As mentioned above, video designers are typically concerned with two metrics: PSNR and encoded bitrate, typically measured in Mb/s. We therefore use these two values as the output abstraction for the video encoder. To compute the distortion, we measure the change in both PSNR and Mb/s as a percentage of these values from the original encoder. We combine the distortions of the two values by taking the weighted average. For this example, we weight PSNR and Mb/s equally.

SpeedPress perforates the `pixel_satd_wxh()` function as follows. It first compiles the encoder with profiling instrumentation that measures the execution time and output quality. It next perforates different code portions and runs the resulting perforated application on representative inputs to record the effect of the perforation on performance and distortion. SpeedPress uses *loop perforation* as its code perforation mechanism (see Section 3). In this example, the compiler perforates the outer loop (over `i`) by generating code that skips every other iteration of the loop. The inner loop (over `j`) is perforated in the same manner. The speedup and distortion of each of these perforations is measured individually by the compiler.

Perforation	Speedup	Distortion
outer loop	1.457	3.65%
inner loop	1.457	4.66%

Table 1: Results of code perforation applied to the SATD function.

Table 1 shows the results of applying perforation to the SATD function in the x264 implementation [32] of H.264 video encoding for a high-definition, 1080p video sequence. In this case, each loop is perforated by skipping every other execution (a *perforation rate* of 50%). The table shows how the speedup of the perforated version compares to the original. In addition, it shows how loop perforation affects distortion. Perforating the outer loop produces a 46 % increase in speed and a distortion of 3.65 %. In this case, the PSNR is reduced by less than 0.4 dB while the impact on Mb/s is 6.70%. Perforating the inner loop also results in a 46 % increase in speed and a similar reduction in PSNR while increasing Mb/s by 8.85%.

Of course, the loops in this function are far from the only loops in the application amenable to perforation. The compiler attempts to perforate each loop. If perforating a loop does not yield speedup, causes unacceptable distortion or causes the program to crash then that loop is not considered as a candidate for perforation. At the end of this process, the compiler has a complete set of speedup and distortion numbers for each candidate loop. We have found that the compiler is often able to discover a number of loops that can be perforated to provide varying performance/accuracy tradeoffs. The candidate loops are ordered by their *score* (see Section 3), which is based on the speedup and distortion caused by perforating that loop. The loops shown in Table 1 represent the two loops found to have the highest scores for x264.

Given a bound on acceptable distortion, the compiler uses the scores of individual loops to find a set of loops which maximizes speedup for that bound. For example, suppose a distortion of 10% is acceptable for x264. In that case the compiler determines a set of loops to perforate that maximizes speedup while keeping distortion below the 10% bound.

To determine the set of loops that provides maximum speedup for a given bound, SpeedPress starts by perforating the loop found to have the highest score during its initial search. SpeedPress then perforates additional loops prioritizing those that have the largest scores. As each loop is added to the set of perforated loops, SpeedPress measures the cumulative speedup and distortion. Loops are added until the maximum allowable distortion is surpassed, at which point the last loop, which pushed the distortion over the acceptable bound, is removed. The remaining set of loops represents the set of perforations which maximize speedup for the given distortion bound.

For x264 and a distortion bound of 10%, SpeedPress finds the two loops in Table 1 and 5 additional loops (for a total of 7 loops) to perforate such that the total speedup is over 2× while the distortion is 9.51%. This distortion is due to a 0.5dB decrease in PSNR (which is just at the widely accepted perceptability threshold of 0.5dB) and an 18% increase in Mb/s. The set of loops that the compiler perforates in x264 is shown in Table 3 and discussed in greater detail in Section 6.

In addition to providing static performance/accuracy tradeoffs, the SpeedPress compiler supports dynamic code perforation, which allows perforation to be turned on and off while the program executes. The SpeedGuard runtime system combines dynamic code perforation with runtime performance monitoring to enable applications to automatically respond to environmental changes that affect performance. To make use of SpeedGuard the programmer specifies a minimum acceptable performance. If SpeedGuard ever detects a performance drop below that level it can dynamically increase perforation to bring performance back to an acceptable level.

To illustrate the use of the SpeedGuard system, consider x264 running on a processor which allows dynamic frequency scaling. The minimal acceptable performance for x264 is set to thirty frames per second corresponding to real-time speed. If the cooling fan for the processor fails, the operating system might adjust the clock frequency to reduce power and heat and keep the processor from failing. The lower clock frequency will result in lower performance which will be detected by SpeedGuard. If the clock frequency changes from 2.5 to 1.75 GHz, SpeedGuard can compensate by perforating the outer loop of the `pixel_satd_wxh()` function to maintain performance at the cost of some distortion as shown in Table 1. The SpeedGuard system is discussed in greater detail in Section 4, while Section 7 provides detailed discussion of how SpeedGuard allows x264 to respond to both core failures and dynamic changes in processor frequency.

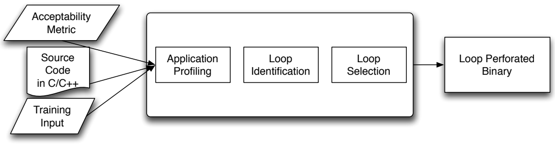

Figure 2: Compiler Framework Overview

3. SpeedPress COMPILER FRAMEWORK

3.1 Overview

Our SpeedPress compiler is built using the LLVM compiler infrastructure [18]. Figure 2 presents an overview of the compiler framework. The compilation process takes as input the application source code, a set of representative inputs, and a user-defined acceptability model. Our evaluation focuses on applications written in C/C++, but since SpeedPress operates on the level of LLVM bitcode, it can support every language for which an LLVM front-end exists (e.g. Fortran, Ada). The representative inputs are used to determine the speedup/distortion tradeoff. The user-specified acceptability model consists of three parts: (1) abstraction of the program output, (2) an accuracy test which measures the effect of code perforation relative to the original (abstracted) output, and (3) the maximum acceptable value of the accuracy test. SpeedPress performs a set of transformation passes that insert instrumentation to perform program profiling, loop identification, and loop selection.

3.2 Profiling

SpeedPress obtains its profiling information as follows. When it compiles the original program for profiled execution, it inserts instrumentation that counts the number of times each basic block is executed. It also inserts instrumentation that maintains a stack of active nested loops. Additional instrumentation is added to count the number of (LLVM bit code) instructions executed in each loop, propagating the instruction counts up the stack of active nested loops so that outermost loops are credited with instructions executed in nested loops. The profile-instrumented version of the program keeps track of the accesses of basic blocks within the loops, the number of loop invocations (times the loop was entered from outside), the number of loop iterations (times the loop body was executed) and dynamic nesting of the loops (parent and children loops) during the execution. SpeedPress uses the resulting instruction counts to prioritize the perforation of loops that execute more instructions in the profiling runs over loops that execute fewer instructions.

3.3 Loop Perforation

Given a loop to perforate, our loop perforation transformation takes as input a percentage of iterations to skip during the execution of the loop, and a perforation strategy. A transformation pass alters the calculation of the loop induction variable to manipulate the number of iterations that a loop will execute. The transformation may also include additional information from the run-time environment. Using the example from Figure 1, the pass conceptually performs the following loop transformation:

```
for( i = 0; i < h; i+=4 ) { /* ... */ }
```

to

```
for( i = 0; i < h; i+=4 ) {
   if (doPerforate(i, environment)) continue;
   //...
}
```

The percentage of non-executed iterations is called the *perforation rate* (*pr*). Depending on the selected perforation rate a different performance/distortion trade-off can be made. For example for a perforation rate $pr = 0.5$, half of the iterations are skipped, for $pr = 0.25$, one quarter of the iterations are skipped, while for $pr = 0.75$, three quarters of the iterations are skipped, i.e. only one quarter of the initial work is carried out.

The compiler supports a range of perforation options, including modulo perforation (which skips or executes every *n*th iteration), truncation perforation (which skips either an initial or final block of iterations), and random perforation (which skips randomly selected iterations at a mean given rate). The actual generated code exploits the characteristics of each specific loop perforation option to generate optimized code for that option.

The current SpeedPress implementation supports both static and dynamic loop perforation. Dynamic loop perforations can be turned on and off by the run-time system, allowing for finer-grained control over the program execution. One example use of dynamic loop perforation is given in Section 4.

3.3.1 Induction variables

SpeedPress perforation operations manipulate loops whose induction variables are in canonical form [19]. It uses an LLVM built-in pass which transforms loop induction variables into a form in which the induction variable `i` has an initial value of 0 and is incremented by 1 in every iteration until `maxvalue` is reached:

```
for ( i = 0; i < maxvalue; i++ ){ /* ... */ }
```

3.3.2 Modulo Perforation

Modulo perforation skips every n-th iteration, *or* executes every n-th iteration. The percentage of skipped iterations is determined by the perforation rate, pr, which is determined using the following formula:

$$pr = \begin{cases} \frac{1}{n} & \text{if every } n\text{-th iteration is skipped} \\ 1 - \frac{1}{n} & \text{if every } n\text{-th iteration is executed} \end{cases}$$

The implementation of modulo perforation considers three cases: (1) large, (2) small, and (3) small where n is power of 2. In the following paragraphs, the implementation for the case when $pr \geq 0.5$ will be referred to as *large* perforation, while the case when $pr < 0.5$ will be referred to as *small* perforation. Additionally, for small perforation, if n is power of 2, a more efficient implementation is available for some computer architectures.

The following examples describe each transformation. The implementation of static perforation is presented first, followed by the implementation details of dynamic perforation.

Large Perforation: For *large* perforation the value of the induction variable increment is changed from 1 to n:

```
for (i = 0; i < maxvalue; i += n) { /* ... */ }
```

Small Perforation: *small* perforation is implemented by adding a new term to the induction variable increment. The goal is to increment the value of the induction variable by 2 when the iteration is to be skipped. The value of the induction variable is incremented by 2 if the remainder of i divided by n is equal to some constant value k, $0 \leq k < n$:

```
for (i = 0; i < maxvalue;
         i = i + 1 + ( i % n == k ? 1:0 ) ) {
//.....
}
```

Small Perforation when n is Power of 2: When n is power of 2 ($n = 2^m$), small perforation uses faster bitwise *and* operations to calculate the remainder of i divided by n, which in this case are lowest m bits of i:

```
for (i = 0; i < maxvalue;
         i = i + 1 + ( i & (n-1) == k ? 1:0 ) ) {
//.....
}
```

Dynamic Perforation

Dynamic perforation allows loop perforations to be turned on and off during the program execution. The runtime system provides a function `doPerforate(LoopId)` to check whether the specific loop should be perforated in the next invocation. `LoopId` is an internal unique loop identifier assigned by the compiler. There are two approaches to implementing dynamic perforation: (1) duplication of loop code and (2) augmenting increments with perforation specific behavior. Loop code duplication involves duplication of (part of) the original loop body, and subsequent modification of the induction variable increment, as outlined for the static perforation case.

Augmented increments, on the other hand, are inserted in the original loop, and are activated only if the loop is to be perforated. To decrease the overhead of the augmented perforation checks, the call to the `doPerforate` function for all nested loops is located in the preheader of the topmost loop, when loops belong to the same function. Additionally, the compiler makes the list of loops that can be dynamically perforated available to the runtime. In the following paragraphs, we discuss the details of augmented increments, and how they modify the original loops.

The dynamic implementation of *large* perforation assigns the value of `increment` based on the perforation check:

```
int increment = doPerforate(loopId)? n : 1

for (i = 0; i < maxvalue; i += increment) { /* .... */ }
```

The dynamic implementation of *small* perforation controls the iteration increment by assigning the appropriate value of the remainder based on the perforation check. It utilizes the property that the value of the remainder must be less than the value of the divisor:

```
int remainder = doPerforate(loopId)? k : n;

for (i = 0; i < maxvalue;
         i = i + 1 + ( i % n == remainder ? 1:0 ) ) {

//.....
}
```

Similarly, *small* perforation for $n = 2^m$ utilizes the property that the result of a bitwise *and* with an m-bit number cannot exceed m bits:

```
int remainder  = doPerforate(loopId)? k : n

for (i = 0; i < maxvalue;
         i = i + 1 + ( i & (n-1) == remainder ? 1:0 ) ) {
//.....
}
```

3.3.3 Truncation Perforation

Truncation perforation skips iterations at the beginning or at the end of the loop execution. The iteration count of the perforated loop is equal to $(1 - pr) \cdot maxvalue$. Discarding iterations at the beginning of the loop involves initialization of the induction variable `i` to `i = pr * maxvalue` where `maxvalue` is known before the loop invocation and is not changed during the loop's execution. If dynamic perforation is used, the induction variable i is initialized based on the result of `doPerforate()`. The example of the loop is:

```
for (i = pr * maxvalue; i < maxvalue; i++) { /* ... */ }
```

Perforating iterations at the end of the loop accomplishes earlier exit from the loop. This is implemented by decreasing the loop condition bound. The new condition becomes `i < (1 - pr) * maxvalue`:

```
for (i = 0; i < (1 - pr) * maxvalue; i++) { /* ... */ }
```

If `maxvalue` is not modified from within the loop body, the new condition can be precomputed. Otherwise, it needs to be checked in every iteration. Instead of performing floating point multiplication, which may be expensive on some architectures, it is possible to represent the rational number $1 - pr$ as p/q, where p and q are natural numbers. Then, the condition can be represented as `q * i < p * maxvalue`. If p or q are powers of 2, shifting may be used instead of multiplication. In case of dynamic perforation, the value of the loop upper bound (or p and q) would be initialized based on the result of the `doPerforate()` function.

3.3.4 Randomized Perforation

Randomized loop perforation skips individual iterations at random, based on a user-specified distribution with mean *pr*:

```
for( i = 0; i < maxvalue; i++ ) {
   if (skipIteration(i, pr)) continue;
   //...
}
```

This type of perforation is the most flexible, but introduces the greatest overhead. It allows the runtime to dynamically control perforation during the execution of the loop body and change the underlying perforation distribution in the course of loop execution. However, the complexity of the `skipIteration()` function may become an issue, due to its frequent execution. It is preferable to apply this technique on loops that have a smaller number of iterations and/or perform more work in each iteration. The call to `skipIteration()` is, in most cases, inlined by the compiler to reduce the call overhead. In the case of dynamic perforation, the loop may be cloned before applying the perforation transformation. Cloning allows the program to use the original version of the loop when the perforation is off and the modified version when the perforation is turned on.

3.3.5 Perforation Mode Discussion

Different perforation options may be more applicable to different types of loops. Modulo perforations are most suitable for the loops that have the work and/or data evenly distributed across the iterations. The loops from Figure 1 are good candidates for modulo perforation. Assuming a perforation rate $pr = 0.5$ the differences that contribute to the final sum are sampled from a "checkerboard" of 4×4 sub-blocks of the macroblock.

Truncation perforations are most suitable for loops that quickly approach an answer in initial iterations and improve the approximation of the final result in latter iterations. For example, from Figure 1, a truncation-perforated program would calculate the sum-of-Hadamard-transformed-difference for only one part (e.g. 8x8 subregion) of the macroblock. This may be appropriate in some cases, but may not be appropriate when a contiguous sub-region of the macroblock is not representative of the entire block. In contrast, truncation perforation can be a good match for simulated annealing programs where the final iterations of the perforated loop refine an approximate answer generated in earlier iterations.

Finally, randomized perforation can be used interchangeably with both modulo and truncation perforations. However, randomly perforated loops should tolerate the overhead of the internal logic and updates to the state of the random number generator. Additional care must be taken for multithreaded programs, as the shared pseudo-random number generator may become a performance bottleneck.

3.4 Acceptability Model

To measure the effect of loop perforation, SpeedPress requires the user to provide an acceptability model for the program output. This model has two components, the first is an output abstraction, while the second is a distortion metric. As part of the model the user provides a bound on the acceptable distortion.

3.4.1 Output Abstraction

The output abstraction is a mapping from a program's specific output to a measurable numerical value or values. In the example in Section 2, the output abstraction consists of the peak signal-to-noise ratio and bitrate.

Creating a program output abstraction is a straightforward process for users with basic knowledge of an application. Without prior knowledge of the PARSEC benchmark applications, we were able to produce output abstractions for each examined application in a short time.

3.4.2 Distortion metric

To evaluate the effect of loop perforation on program output we use an accuracy test based on the relative scaled difference between selected outputs from the original and perforated executions. Specifically, we assume the program output abstraction produces an output of the form $o_1, \ldots, o_m$, where each output component o_i is a number.

Given an output $o_1, \ldots, o_m$ from an unmodified execution and an output $\hat{o}_1, \ldots, \hat{o}_m$ from a perforated execution, the following quantity d, which we call the *distortion*, measures the accuracy of the output from the perforated execution:

$$d = \frac{1}{m} \sum_{i=1}^{m} \left| \frac{o_i - \hat{o}_i}{o_i} \right|$$

The closer the distortion d is to zero, the less the perforated execution distorts the output. Because each difference is scaled by the corresponding output component, distortions from different executions and inputs can be compared. By default the distortion equation weighs each component equally, but it is possible to modify the equation to weigh some components more heavily. For more on distortion see [22].

3.4.3 Bias Definition and Use

The distortion measures the absolute error induced by loop perforation. It is also sometimes useful to consider whether there is any systematic direction to the error. To measure systematic error introduced through loop perforation we use the *bias* [22] metric:

$$b = \frac{1}{m} \sum_{i=1}^{m} \frac{o_i - \hat{o}_i}{o_i}$$

Note that this is the same formula as the distortion with the exception that it preserves the sign of the summands. Errors with different signs may therefore cancel each other out in the computation of the bias instead of accumulating as for the distortion.

If there is a systematic bias, it may be possible to compensate for the bias to obtain a more accurate result. Consider, for example, the special case of a program with a single output component o. If we know that bias at a certain is b, we can simply divide the observed output $\hat{o}$ by $(1 - b)$ to obtain an estimate of the correct output whose expected distortion is 0.

3.5 Loop Selection

The goal of the loop selection algorithm is to find the set of loops that can be perforated to produce the highest performance increase for the lowest output distortion value, given the maximal acceptable distortion and desired perforation rate. The algorithm for loop selection performs the following steps:

- Identification of candidate loops for perforation.
- Measurement of performance and distortion for each candidate loop.
- For each input, the discovery of the loop set that maximizes performance for a specified distortion bound.
- Selection of the loop set that provides the best results for all training inputs.

```
LoopSelection (program, inputs, maxDist)
   candidateLoops = {}
   scores = {}

   for i in inputs
      candidateLoops[i] = performProfiling(program, i)
      for each l in candidateLoops[i]
         scores[i][l] = assignInitialLoopScore(l)
      filterProfiledLoops(candidateLoops[i])

      for l in candidateLoops
          spdup, dist = perforateLoopSet(program, {l}, i)
          scores[i][l] = updateScore(spdup, dist, scores[i][l])
      filterSingleExampleLoops(candidateLoops[i], scores[i], maxDist)

   candidateLoops, scores =
      mergeLoops(candidateLoops[*], scores[*])

   if size(candidateLoops) == 0
       return {}

   candidateLoopSets = {}
   for i in inputs
      candidateLoopSets[i] =
         selectLoopSet(program, candidateLoops, scores, i, maxDist)

   loopsToPerforate =
       findBestLoopSet(program, candidateLoopSets, inputs)
   return loopsToPerforate
```

Figure 3: Loop selection algorithm pseudocode

The pseudocode of the SpeedPress loop selection algorithm is given in Figure 3. The steps of the algorithm are described in the following sections.

3.5.1 Identification of Candidate Loops

Initially, all loops are candidates for perforation. The algorithm invokes the profile-instrumented program, described in Section 3.2 on all training inputs in order to find candidate loops for perforation. Each loop is given a score according to its effect on the program execution time and the number of invocations. The score is calculated based on the normalized values of instruction count and invocation number. The loops that have only a minor contribution to the program execution time, an unsatisfactory number of iterations/invocations, or that cannot be instrumented are filtered from the candidates list.

3.5.2 Individual Loop Performance and Distortion

The algorithm perforates each candidate loop in isolation and observes the influence of the perforation on the speedup and distortion. The loop is statically perforated with a predefined perforation rate. The pseudocode for the execution of loop perforation is given in Figure 4. After the execution of the instrumented program SpeedPress uses the acceptability model to calculate the distortion from the expected output.

An extension of the algorithm tries different perforation rates for individual loops in order to fit them within the bound. If the speedup contribution of the added loop is not positive, the step will be repeated with a higher perforation rate for the loop. If the distortion introduced by adding a loop is greater than allowed, the perforation rate of the loop may be decreased. Due to the potentially large number of additional training runs, this extension is primarily used for loops that have a greater influence on execution time.

The score for each loop is updated based on the measured speedup and negative distortion from the performed executions and calculated as a weighted harmonic mean. Using a weighted harmonic mean allows a user to prioritize loops either by greater speedup (although the distortion may be larger, leading to fewer perforated loops), or by smaller distortion, at the expense of the speedup (leading to more perforated loops, each having lesser influence on program speedup).

```
perforateLoopSet(program, loopSet, input)
   program' = instrumentLoops(loopSets)

   time, output = execute(program, input)
   time', output' = execute(program', input)

   abstrOut = abstractOutput(output)
   abstrOut = abstractOutput(output')

   speedup = calcluateSpeedup(time, time')
   distortion = calculateDistortion(abstrOut, abstrOut')

   return speedup, distortion
```

Figure 4: `perforateLoopSet` pseudocode

The current implementation uses only one, predefined perforation strategy for all loops. It is straightforward to extend the algorithm to automatically select a perforation strategy (e.g. modulo, or truncate) based on profiling information and the additional training executions.

It is possible that a program with perforated loops will terminate unexpectedly or hang during loop evaluation. If the program terminates due to error (e.g. segmentation fault), the distortion is set to 100%, disqualifying the loop from further consideration. If the program is not responsive for a time greater than the execution of the reference version, it is terminated, and the speedup set to 0, also disqualifying the loop. Loops that do not increase the performance and loops that cause distortion greater than the maximum bound specified by the user are also removed from the candidates list.

After assigning scores for all individual loops, the algorithm merges the results from multiple examples. The loop scores are averaged over all inputs. Only the loops that have positive scores for all inputs remain as candidates. If there are no such loops, the selection algorithm terminates and returns the empty set.

At this point, the algorithm has finished its initial exploration of candidate loops for perforation. The following steps search for a set of candidate loops that, when perforated simultaneously, provide maximum speedup while keeping distortion below the acceptable bound.

3.5.3 Discovery of Loop Sets with Acceptable Distortion

The next step is to combine loops with high individual scores on all training inputs and observe their joint influence on program execution. Note that the distortions and speedups of programs with multiple perforated loops may not be linear in terms of the individual perforated loop results (because of potentially complex interactions between loops, including nesting, work dispatching, etc.). This step is executed separately for each training input. Pseudocode for multiple loop selection is given in Figure 5.

The algorithm maintains a set of loops that can be perforated without exceeding the maximal acceptable distortion bound (`maxDist`) selected by the user. At each step, it tries the loop with the highest individual score, and executes the program where all the loops from the set and the new loop are perforated. If the performance increases, and the distortion is smaller than the maximum allowable, the loop is added to the set of perforated loops.

3.5.4 Selection of the Best Performing Loop Set

Finally, the loop sets from all training inputs are compared, as shown in Figure 6. The best loop sets for each input are executed

```
selectLoopSet(program, candidateLoops, scores, input, maxDist)

   loopQueue = sortLoopsByScore(candidateLoops, scores)

   LoopSet = {}
   cummulativeSpeedup = 1
   while loopQueue is not empty
      tryLoop = loopQueue.remove()
      trySet = LoopSet U {tryLoop}
      speedup, distortion = runPerforation(trySet, input)

      if speedup > cummulativeSpeedup and distortion < maxDist
         loopSet = trySet
         cummulativeSpeedup = speedup

   return LoopSet
```

Figure 5: `selectLoopSet` pseudocode

on other inputs. The score for each loop set is derived as a statistic (e.g. minimum or mean) of the scores of executing the loop sets on all inputs. A loop set that fails to terminate normally on some input is excluded from the set of candidates. The loop set with the best score on all training inputs is returned as the final loop selection.

```
findBestLoopSet(program, loopSets, inputs)

   for each ls in loopSets
      for each i in inputs
         speedup, distortion =
                 perforateLoopSet(program, ls, i)
         score[ls][i] = assignScore(speedup, distortion)

      score[loopSet] = scoreFinal(score[ls][*])

   return argmax(score)
```

Figure 6: `findBestLoopSet` pseudocode

The order of the loops is important if SpeedPress performs dynamic perforation. The set of loops in the loop set may be ordered by their individual scores. The compiler encodes the information about the speedup and distortion of perforated loops and makes it available to the runtime subsystem. If the compiler performs static perforation of the program, the set of loops is unordered, since all loops are perforated throughout the program's execution.

4. SpeedGuard RUNTIME SYSTEM

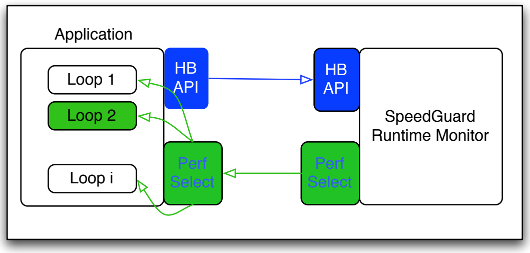

Figure 7: SpeedGuard runtime system

This section describes the SpeedGuard runtime system for applying Code Perforation for fault tolerance. Figure 7 shows an overview of the SpeedGuard system, which consists of three components.

SpeedGuard uses the Application Heartbeats API to monitor the dynamic performance of the application [16]. The loop perforation selector enables dynamic loop perforation selection. The SpeedGuard monitor orchestrates responses to events (such as failures, recoveries, changes in the load, and frequency scaling) by dynamically selecting loops to perforate.

4.1 Application Heartbeats

The Application Heartbeats API provides a standardized way for an application to report both its performance goals and its dynamic performance. Using this API, an application registers a *heartbeat* at some interval. The performance of the application is then measured in terms of its heart rate. Additional functions allow the application to specify its goals in terms of a minimum and maximum heart rate. External services, such as the operating system, can query an application's current heart rate as well as its desired minimum and maximum.

The SpeedGuard framework requires that the programmer make use of the Heartbeat API for any managed application. The programmer is responsible for placement of heartbeat calls, establishing the minimum and maximum heart rate as well as the call to register the heartbeat. This procedure typically requires minimal code changes and need only be performed once per application.

Adding the appropriate heartbeat calls is typically straightforward given a basic knowledge of the application. The PARSEC documentation describes the inputs processed by each benchmark. Given this description, we were able to quickly find the loop that iterates through the input and place the call to register the heartbeat in the loop. At most a half-a-dozen lines of code were required to augment each benchmark with the Heartbeat interface.

Establishing the minimal and maximal heart rate should be easy for anyone who wants to make use of SpeedGuard. Since SpeedGuard is designed to monitor performance and keep it within a specified bound, we assume that the user already has determined the desired performance for the application.

4.2 Dynamic Loop Perforation

The SpeedGuard framework can be incorporated with SpeedPress to support dynamic loop perforation. In this case, the compiler performs dynamic perforation of the loops after finding the set of loops that improve performance while keeping accuracy within an acceptable range. Turning loop perforations on and off dynamically, during the course of program execution, results in performance increase while keeping the accuracy within the acceptable range.

The compiler provides the runtime with a table that contains all information about the dynamically perforated loops. For any loops that SpeedGuard considers perforating, it can look up the expected speedup and the expected distortion. Having this information allows the runtime to make informed decisions in response to dynamic events.

Perforation is initially set to be off for all loops, so the application initially runs as the unmodified version. Additionally, the compiler adds a run-time system that adjusts perforation as described in the next section.

4.3 Runtime Monitor

SpeedGuard is inserted by the compiler to monitor the application's heart rate by making calls to the Heartbeat API. If the heart rate ever falls below the desired minimum, the SpeedGuard system increases the amount of perforation. If performance later increases above the desired maximum, the run-time decreases the amount of perforation. SpeedGuard only perforates loops that have been discovered and profiled by the compiler so their effect on the distortion is known.

The SpeedGuard monitor checks performance at a given interval specified in terms of heartbeats. For example, the monitor may check performance every 20 heartbeats. This interval can be tuned for the needs of a particular deployment of an application. Setting the interval to be small results in more aggressive response to performance changes while a larger interval leads to a more conservative response.

When performance dips below the desired minimum, SpeedGuard computes the percentage difference. With this information, SpeedGuard can determine a loop whose dynamic perforation results in a speedup that is most likely to reclaim the lost performance. For example, if SpeedGuard detects performance 30 % below the desired minimum, it will attempt to dynamically perforate a loop that the compiler previously found to provide a speedup of 1.3.

SpeedGuard supports two modes of operation. In the first, aggressive mode, it attempts to reclaim all lost performance at once. Operating aggressively, the runtime may perforate multiple loops to regain performance. In the second, conservative mode, the runtime will dynamically perforate one loop, observe the change in performance and then perforate additional loops as needed. The aggressive mode tends to have a faster response to performance changes, while the conservative mode tends to have the smallest impact on distortion.

It is possible that SpeedGuard cannot find one set of loops that keeps performance within the desired bounds. In this case, the runtime will continuously raise and lower perforation to keep the average performance in the desired bound. An example of this behavior is shown in Section 7.

4.4 Example

To illustrate how SpeedGuard operates, consider again the video encoder example from Section 2. To make use of this framework, the programmer first modifies the encoder to use the Heartbeat API so that it registers a heartbeat as each frame is encoded. Additionally, the programmer uses the API to specify the desired minimum and maximum heart rate. In this case, the minimum might be thirty beats per second, corresponding to thirty frames per second.

Having added a heartbeat to the application, the programmer then submits it to SpeedPress. The compiler applies perforation and adds the ability to turn this perforation on and off. Additionally, the compiler inserts the calls to the run time system that monitor application performance and adjust the level of perforation.

The resulting encoder can then be deployed. Initially the encoder will run at the desired speed of thirty frames per second. Suppose the fan cooling the system fails and the operating system reduces the processor frequency from 2.5 to 1.6 GHz to reduce power and temperature. This change in frequency will result in a change in the application performance to about 19 frames per second. The change in performance is reflected in the change in heart rate. When the application's heart rate changes, it is detected by SpeedGuard which perforates the inner loop of the SATD function to increase performance by 45 % as shown in Table 1. This will raise performance back to about twenty-seven frames per second. If performance is still not acceptable, the runtime can perforate the outer loop, bringing performance back up to the desired target.

In practice, the compiler finds a much richer set of loops to perforate in a video encoder which gives the run time system more options and more fine-grained control over performance adjustments. See section Section 7 for the results of applying the full system to create a fault tolerant video encoder.

5. EVALUATION METHODOLOGY

This section presents the methodology used to evaluate SpeedPress and its ability to find meaningful performance-accuracy tradeoffs for a wide range of applications. All results are collected using a single core of an Intel x86 server with dual 3.16 GHz Xeon X5460 quad-core processors. Benchmarks are taken from the PARSEC 1.0 benchmark suite because of its focus on capturing a diverse set of emerging workloads [8]. These workloads are designed for the next generation of processor architectures and their high computational load makes them candidates for code perforation as it can reduce this load at the cost of some accuracy loss.

5.1 Benchmarks

We use the following benchmarks in our evaluation. Together, these benchmarks represent a broad range of computations including financial analysis, media processing, engineering, and data mining workloads.

- **x264.** This media application performs H.264 encoding on a video stream. Distortion is calculated using the peak signal-to-noise ratio (PSNR) of the encoded video as well as the bitrate of the encoded video. PSNR is measured using the H.264 reference decoder. For each of these values the perforated output is compared to the original. The distortions for PSNR and bitrate are then averaged. The x264 benchmark includes both assembly and vanilla C implementations of some functions. We use the C implementations to give the compiler a larger set of loops to perforate.

- **streamcluster.** This data mining application solves the online clustering problem. Distortion is calculated using the *BCubed* (B^3) clustering quality metric [2]. The metric calculates the homogeneity and completeness of the clustering generated by the application, based on external class labels for data points. The value of the metric ranges from 0 (bad clustering) to 1 (excellent clustering). The distortion is represented as the scaled difference between the clustering quality of the perforated and original code. If the perforated program performs better than the unmodified version it is considered to have no distortion.

- **swaptions.** This financial analysis application prices a portfolio of swaptions by using Monte Carlo simulation to solve a partial differential equation. Distortion is calculated using the price of the swaption and comparing the price of the perforated application to that calculated by the original. The PARSEC benchmark only uses a single set of parameters so all swaptions have the same value. To obtain a more realistic computation, the input parameters were altered so that the underlying interest rate of the swaption can vary from 0 – 10 %.

- **canneal.** This engineering application uses simulated annealing to minimize the routing cost of microchip design. Distortion is calculated using the routing cost and comparing the cost of the perforated version to the cost of the original.

- **blackscholes.** This financial analysis application computes the price of a portfolio of European options by solving a partial differential equation. The distortion is calculated by comparing the price of options determined by the perforated application to the price generated by the original application.

- **bodytrack.** This computer vision application tracks a human's movement through a scene using an annealed particle filter. Distortion is calculated using a series of vectors that represents the changing configurations of the body being tracked. Distortion is measured by computing the relative mean squared error (RelMSE) for each vector by comparing the vectors generated by the perforated version to those produced by the original. The RelMSE for each vector is then divided by the magnitude of the vector produced by the original and these values are averaged for all vectors.

In addition to these benchmarks, the PARSEC benchmark suite contains the following benchmarks: facesim, dedup, fluidanimate, ferret, freqmine, and vips. We do not include freqmine and vips because these benchmarks do not successfully compile with the LLVM compiler. We do not include dedup and fluidanimate because these applications produce complex binary output files. Because we were unable to decipher the meaning of these files, we were unable to develop meaningful acceptability models and distortion metrics. We do not include facesim because it does not produce any output at all (except timing information).

Finally, we do not include a detailed evaluation of ferret because all of the time-intensive loops in this application fall into one of two categories: either perforating the loop causes unacceptable output distortion or the loop is part of a filtering phase that coalesces image segments for use during a subsequent evaluation phase. Because perforating such loops increases the number of image segments considered during the evaluation phase, this process actually decreases the overall performance. The initial exploration of the performance/accuracy trade off space indicates that loop perforation is unable to acceptably increase the performance of this application.

5.2 Perforated Executions

Each PARSEC application comes with three training inputs and one native input. The training inputs are provided to support activities (such as compiler optimizations that use dynamic profiling information) that exploit information about the run-time behavior of the application prior to building the production version. The native inputs are designed to enable the evaluation of the application on production inputs. We use the training inputs to explore the performance/accuracy tradeoff space and select loops to perforate that provide the best combination of performance and accuracy (see Section 6). We report performance and accuracy results from executions that process the native inputs. These inputs were not used as representative inputs during the exploration of the performance/accuracy tradeoff space.

In all experiments except canneal, we set the compiler perforation rate to 0.5 (i.e., each perforated loop skips half of the loop iterations). The modulo perforation strategy is used for all loops. If a loop contributes less than 1% to the total execution time, the compiler never perforates the loop (and does not evaluate the effect of perforating the loop during the exploration of the performance/accuracy tradeoff space). Because canneal delivers a better combination of performance and accuracy at a higher loop perforation rate, we use a perforation rate of 0.97 for this benchmark.

For some of the applications the provided datasets are not always appropriate for providing the full coverage needed to train the compiler. All training inputs for x264 use the same video in different resolutions. To avoid potential over-fitting, we perform the training on different video with the same resolution as the native input (1080p). For bodytrack, the training datasets contain at most 4 frames, which is insufficient for the training. We perform the training using the first 20% (52 frames) of the native data set. In streamcluster, points are drawn from a uniform distribution across the input space – since the data is not cluster-able, it is impossible to assess the effect of code perforation. We therefore use an existing clustering evaluation data set `covtype`[1] for testing. We also create our own synthetic training inputs for this application. These training inputs have a predefined number of centers, with other points normally distributed around the centers.

6. EVALUATION OF SpeedPress

This section presents an evaluation of the SpeedPress perforating compiler applied to the six PARSEC benchmarks described in Section 5.1. The section begins by discussing general trends and then discusses each benchmark individually.

6.1 General Trends

Figures 8–13 present, for each application, the normalized performance (left y axis) and distortion (right y axis) as a function of the acceptable distortion bound. The normalized performance is computed as the speedup — the execution time of the original version divided by the execution time of the perforated version. The distortion is presented as a percentage. For each application, the test input is the native input from the PARSEC benchmark suite.

In general, both the performance and distortion increase as the acceptable distortion bound increases. Several points exhibit non-monotonic behavior (the performance decreases as the acceptable distortion bound increases). We attribute these anomalies to differences in the execution characteristics of the loops in the application when run on the training versus native inputs. For others, increasing the distortion bound does not enable the perforation of additional loops, so the performance stays the same as the acceptable distortion bound increases.

For all benchmarks, SpeedPress is able to provide at least 2× speedup for a maximum distortion bound of 15 %, and several of the benchmarks can achieve 3× or greater speedup. In addition to finding large speedups, SpeedPress keeps the distortion close to the acceptable limit used during training. These results demonstrate SpeedPress's ability to deliver significant performance gains while keeping the distortion within a (small) given bound. Furthermore, these results show that SpeedPress can successfully increase the performance of a range of computations, including financial analysis, media processing, computer vision, and engineering computations.

Table 3 shows the data that SpeedPress collected during training for each of the six benchmarks. For each benchmark, the table shows the function where the loop was found, the loop's individual effect on distortion and speedup, and the cumulative effect of distortion and speedup when perforating that loop and every loop above it in the table. This table demonstrates the importance of searching for multiple loops to perforate, as for four of the six benchmarks, multiple loops are perforated to achieve the best speedup.

6.2 Individual Benchmark Evaluation

This section presents a detailed analysis of the performance/accuracy tradeoffs for each of the examined benchmarks. For each benchmark we discuss the loops perforated by SpeedPress during the training run with the 10 % distortion bound, discuss the impact of distortion on the usability of the application, and discuss scenarios in which perforated execution may be preferable.

6.2.1 x264

Figure 8 shows the space of performance/accuracy tradeoffs SpeedPress finds for the x264 benchmark. As described in Section 2, SpeedPress discovered considerable speedup opportunities

[1] Publicly available at UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/)

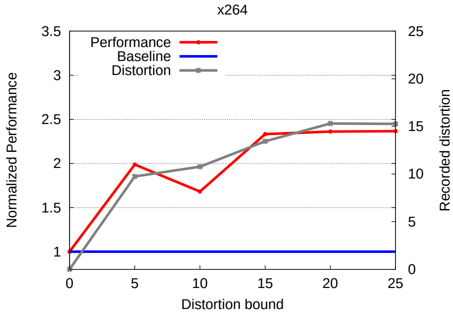

Figure 8: x264

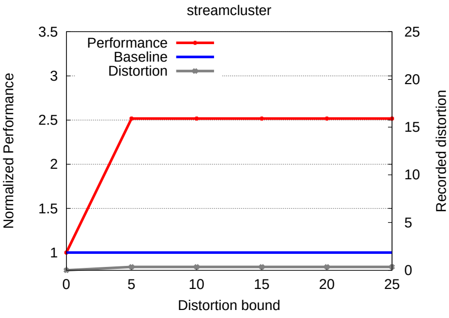

Figure 9: streamcluster

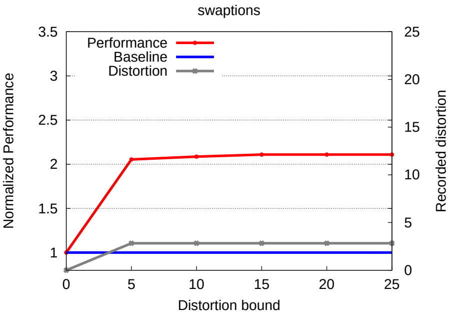

Figure 10: swaptions

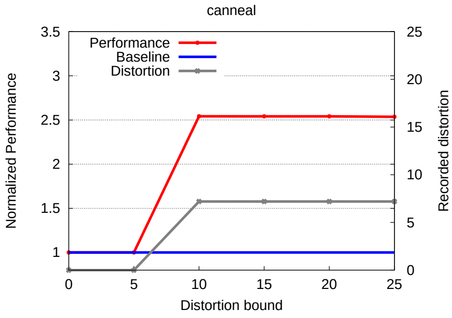

Figure 11: canneal

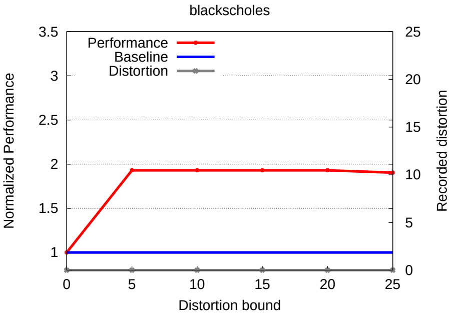

Figure 12: blackscholes

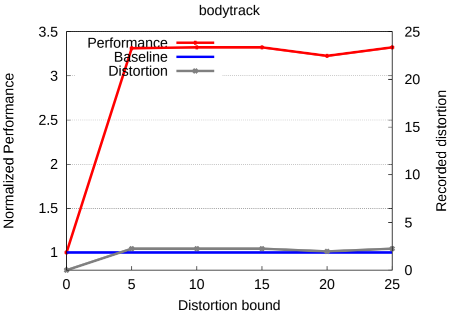

Figure 13: bodytrack

with low output distortion by perforating loops in x264's motion estimation functions. Motion estimation is a computationally expensive component of x264's total computation and one function in particular, `pixel_satd_wxh()`, contributes to over 40% of the total runtime. This function was described in Section 2. Clearly, perforating this loop will improve performance.

Table 3 shows the loops that SpeedPress perforates for x264 during training using a 10% distortion bound. All seven loops are functions used for motion estimation. The `pixel_satd_wxh()` function is the same one described in the example from Section 2. The H.264 standard allows motion estimation to be performed to quarter-pixel accuracy (using interpolation to generate these *sub-pixel* values). The `refine_subpel()` function manages the process of sub-pixel motion estimation and perforating the loop in this function reduces the number of locations that are searched for sub-pixel matches. The `pixel_sad_8x8()` is similar to the SATD function discussed in the example section, except that this function computes the sum-of-absolute-differences for 8×8 regions of the reference and current frames without using a Hadamard transform. The function `x264_me_search_ref()` manages the entire process of motion estimation, and perforating this loop reduces the number of locations searched during this phase of computation. Finally, the function `pixel_sad_8x16()` is similar to `pixel_sad_8x8()` except it computes the sum-of-absolute-differences on 8×16 regions of the current and reference frames.

For the native input, SpeedPress is able to achieve a 2× speedup on x264 with less than 10% distortion. In fact, the changes to image quality are likely to be imperceptible to humans as the PSNR of perforated x264 is within .3 dB of the unperforated version when processing the native input. The major contributor to distortion is file size; the perforated version increases the size of the file by 18% for the native input.

There are several areas where using a perforated version of x264 may be a preferable to the original. One example is transcoding, where one might want to convert an HD video to be viewed on a personal electronic device. Here the speed of conversion may be more important than saving memory. Also, real-time encoders may prefer to run faster and meet performance deadlines if there is sufficient bandwidth available to handle the increase in file size. All these concerns can be expressed using the acceptability model.

6.2.2 *Further Encoding Studies*

Given the importance of motion estimation in video encoding, it is surprising that it is possible to perforate the loops that govern motion estimation and still produce an acceptable result. These results suggest two further studies. The first examines the effects on the speedup and distortion if we simply do no motion estimation. The second examines the effect of doing motion estimation with a 100% perforation rate for the nested loop structure in the `pixel_satd_wxh()` function (which, in effect, eliminates the execution of the loop). We perform both studies using the native input. Table 2 summarizes the results of these additional studies. For reference, the table repeats the results of the 50% perforation rate applied to both loops in `pixel_satd_wxh()`.

Experiment	Speedup	Distortion
No motion estimation	6.80	178.0%
pr = 100% for `pixel_satd_wxh()`	1.75	116.0%
pr = 50% for `pixel_satd_wxh()`	1.67	9.67%

Table 2: Additional studies examining perforation in x264.

Table 2 shows the speedup and distortion when we modify the encoder to eliminate motion estimation. This implementation of the encoder does not attempt to exploit temporal redundancy. Instead, it encodes each frame using only spatial redundancy (or redundancy found within a single frame). Such an encoder is known as an I-frame (or intra-coded frame) only encoder. As shown in Table 2, eliminating motion estimation provides a substantial speedup, but this speedup comes at the cost of 178% distortion. In this case, the distortion is due entirely to the 355% increase in the size of the encoded video.

Table 2 also presents the speedup and distortion for our second study, when we apply a 100% perforation rate to the loops in the `pixel_satd_wxh()` function. This implementation of the encoder will still perform motion estimation, but due to the complete elimination of the loops in the macroblock comparison function this function will always return zero. Returning zero causes any attempted match for a macroblock to be viewed as a perfect match. As shown in Table 2, the 100% perforation rate provides a speedup of 1.75, which is better than that found using a 50% perforation rate; however, this speedup comes with 116% distortion. This distortion is due to a 233 % increase in the size of the encoded video and a .3 dB loss of PSNR.

These studies show that eliminating motion estimation entirely, or even eliminating a key part of it, produces unacceptable results. These conclusions by themselves are not surprising as motion estimation is a key feature of video encoding. However, it is surprising that code perforation can skip a significant amount of the motion estimation computation while still producing encoded video with acceptable distortion. These studies highlight the ability of SpeedPress to automatically find non-trivial parts of a computation to skip while still producing acceptable output.

6.2.3 streamcluster

Figure 9 shows the space of performance/accuracy tradeoffs SpeedPress finds for the streamcluster benchmark. This benchmark solves the online clustering problem by partitioning a set of points such that each is assigned to a group with the closest mean. SpeedPress finds two loops to perforate for the streamcluster benchmark as shown in Table 3. The first loop, in function `pFl()`, estimates the cost of opening a new cluster center. Perforating this loop allows the application to make a less accurate estimate of this cost more quickly. The second loop, in function `dist()`, calculates the distance between two points. Perforating this loop effectively allows the application to estimate the distance by treating the points as if they had lower dimensionality. SpeedPress achieves a speedup of 2.5x with a distortion of 0.35%.

There are several applications for which the increased speed of the perforated streamcluster might be preferable. For example, consider the problem of performing data mining on a stream of network traffic. In this scenario it might be more important to get an approximation of the clustering while maintaining performance that keeps up with the rate of network traffic. Using perforation allows this tradeoff. Another potential use is to quickly assess some unknown initial clustering parameters, such as an estimation of the minimal and maximal number of clusters. After this initial assessment, a more accurate, but slower version of the code could be used.

6.2.4 swaptions

Figure 10 shows the space of performance/accuracy tradeoffs SpeedPress finds for the swaptions benchmark. This benchmark computes the price of a portfolio of swaptions using Monte Carlo simulation. swaptions is the only benchmark of the six which shows significant bias, and correcting that bias is important for allowing the compiler to achieve significant speedup. When SpeedPress accounts for bias, it perforates three loops in the `HJM_Swaption_Blocking()` function. This function is the where most of the time is spent in the program and these loops govern how many Monte Carlo simulations are used. The function `HJM_SimPath_Forward_Blocking()` computes and stores the results of one simulation. Perforating these four loops results in a speedup of 2×, while the bias adjusted output has a distortion of only 2 %.

The ability of the perforated and bias adjusted swaptions to approximate the true result in a fraction of the time has several applications. For example, an application could use perforation to speculatively price swaptions and then slowly compute the exact price later as a monitoring step. In addition, in a very volatile situation it may be preferable to quickly achieve an estimate of the swaption price and act on that price immediately rather than waiting for a slower, but more accurate result.

6.2.5 canneal

Figure 11 shows the space of performance/accuracy tradeoffs SpeedPress finds for the canneal benchmark. This benchmark performs the place-and-route function on a processor netlist using simulated annealing. As shown in Table 3, SpeedPress finds only one significant loop to perforate for this application, in the `run()` method of the `annealer_thread` class. This loop is a `for` loop nested in a `while` loop. The `while` loop checks a termination condition and if it does not terminate, then the for loop attempts a number of moves to improve the routing cost. Perforating this `for` loop causes less work to be done between checking the termination condition. Checking the termination condition more often allows the program to exit sooner in the case where it has reached a point of diminishing returns — the point when moves are more likely to be rejected than accepted. In exchange for the earlier termination, the perforated application attempts fewer moves at this point of diminishing returns and thus misses some additional moves that further reduce cost.

While the training run estimated the speedup for canneal to be 1.3× with a 10 % distortion bound, the actual speedup measured using the test data set shows a speedup of over 2.5× for the same distortion bound. We attribute this difference to the much greater complexity of the test data set compared to that used for training. The native input netlist is over six times larger than the simlarge input. The larger complexity of the input means that the unperforated program spends a longer time operating past the point of diminishing returns.

The perforated canneal could be useful for rapid prototyping, saving the more time consuming full version for the final step. Chip layout is a time-consuming part of processor design and a perforated version of such an application allows engineers to explore different designs quickly.

6.2.6 blackscholes

Figure 12 shows the space of performance/accuracy tradeoffs SpeedPress finds for the blackscholes benchmark. As shown in the figure, both the speedup and distortion curves are flat. This is because SpeedPress only finds one loop to perforate. This is the outer loop of the `main()` function, as shown in Table 3. SpeedPress determines that this single loop can be perforated to provide a speedup of almost 2 with no distortion.

The fact that SpeedPress is able to double the speed of the application without affecting the result is surprising. Closer inspection of this loop reveals it is superfluous as it does not affect the result. Rather, this loop appears to have been added to the benchmark solely to increase the workload and causes the same results to be computed repeatedly. This benchmark demonstrates how SpeedPress can be used to find redundant computation.

6.2.7 bodytrack

Figure 13 shows the space of performance/accuracy tradeoffs SpeedPress finds for the bodytrack benchmark. As shown in the figure, SpeedPress is able to find good speedups for this benchmark

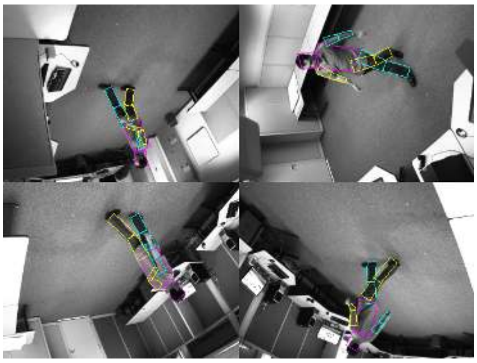

Figure 14: Reference frame.

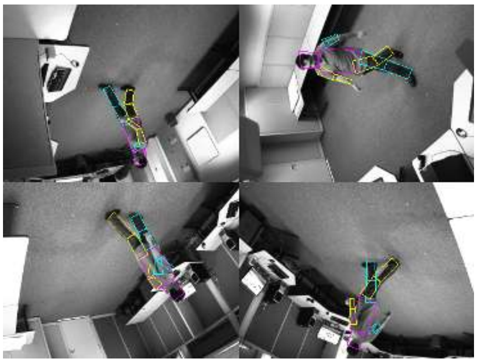

Figure 15: Output of bodytrack in the presence of core failures with perforation.

with very little distortion. bodytrack uses an annealed particle filter and Monte Carlo simulation to track a human through a scene and SpeedPress finds several loops to perforate within the annealing and Monte Carlo steps. The loop with the biggest impact is the one which updates the application's internal model of the body's current pose. This loop is a good candidate for perforation due to its significant contribution to program runtime and because it uses particle annealing and Monte Carlo methods to approximate the position of the body at each time step. Additionally, the compiler finds several other loops that compute measurements of various image features and determines that they can be perforated without significant loss in accuracy.

Table 3 shows that SpeedPress is able to find a number of loops to perforate, and the total effect of perforating these loops results in a speedup of more than 3× while keeping distortion below 2%. The loop in the `Update()` method of the `ParticleFilter` class is the one mentioned above that controls the Monte Carlo and particle annealing steps. For each new frame, bodytrack computes the position of the body and the likelihood that the new position is correct given the old position. The `ImageErrorInside()` function contributes to this calculation. `GetObservation()` reads the data for a new frame and performs some image processing including edge detection and creation of binary images. Perforating `GetObservation()` means that the application uses less data for subsequent computation. The next four functions for bodytrack listed in Table 3 all contribute to determining the location of various body parts in the current frame. The `OutputBMP()` function writes an image illustrating the current position of the body. Perforating this function does not affect distortion because our output abstraction for bodytrack only includes the numerical representation of the body's position and not the output image[2]. The last three loops listed in the table perform a one-dimensional filter on rows of the image.

As shown in Figure 13, despite considerable performance gains, distortion levels remain low. This is due to the fact that the distortion metric is influenced heavily by a small number of relatively large magnitude components of the vector which describes the body's configuration. The large components represent the chest and head of the body being tracked. The perforated version of bodytrack is able to identify and track the head and chest with high accuracy at the cost of reduced accuracy for the arms and legs. The degree to which the application miscalculates the position of the arms and legs is proportional to the amount of perforation. Figure 15 shows how perforation affects the output of bodytrack for our experiments on fault tolerance. Section 7 contains detailed discussion of these figures.

The perforated version of bodytrack may be appropriate for several machine vision applications where quick understanding of the general location of a person is more important than determining the exact configuration of their appendages. An example of such a scenario is an autonomous car that needs to quickly determine whether or not a person is on a course likely to result in collision. Also security cameras may need to quickly determine whether or not a person has entered a scene without regard to their exact configuration.

7. EVALUATION OF SpeedGuard

This section presents several experiments illustrating how the SpeedGuard runtime system can use the performance/accuracy trade-offs discovered by SpeedPress to provide robustness in the face of faults. In the first experiment, the run-time responds to simulated core failures. In the second experiment, the run-time responds to a dynamic change in clock speed. We focus on our fault tolerance experiments on x264 and bodytrack because they are the largest (in lines-of-code) and most complicated of the six benchmarks and therefore provide the most realistic scenarios in which to evaluate SpeedGuard.

For these experiments, we use a different video as input to x264. The native input video of PARSEC has regions of varying difficulty making it difficult to separate performance changes due to the environment from changes due to the input. For this reason we use a video which is, on average, more difficult to encode, but the difficulty does not vary much from frame to frame. For bodytrack, we use the native input in all fault tolerance experiments.

We measure the overhead of SpeedGuard by comparing the speed of the benchmarks with the run-time enabled to the speed with no run-time system. We find that SpeedGuard contributes less than 1% to the execution time and conclude that the overhead of the system is insignificant.

7.1 Core Failure

In the first experiment, SpeedGuard responds to simulated core failures. Parallel versions of x264 and bodytrack are run on eight cores of an Intel Xeon X5460 dual quad-core processor. At a given point in the computation core failure is simulated by restricting the operating system from scheduling the benchmark on three of the eight cores. The target performance for both programs is communicated to SpeedGuard using the Heartbeat API. Both applications request a minimal heart rate between 90% and 115% of what they achieve in a system with no failures.

Figures 16 and 17 show the results of the core failure experiments for each of the benchmarks. These figures show the dynamic

[2] For experiments on fault tolerance, we disable perforation in this loop to have visual confirmation that the output is acceptable.

x264				
	Individual		Cumulative	
Function	Distortion	Speedup	Distortion	Speedup
pixel_satd_wxh, outer	3.65%	1.457	3.65%	1.460
pixel_satd_wxh, inner	4.66%	1.457	9.67%	1.672
refine_subpel	0.05%	1.098	9.83%	1.789
pixel_sad_8x8, outer	0.01%	1.067	9.96%	1.929
pixel_sad_8x8, inner	0.03%	1.060	9.94%	1.986
x264_me_search_ref	0.21%	1.014	9.51%	2.054
pixel_sad_8x16, outer	0.02%	1.005	9.53%	2.058

streamcluster				
	Individual		Cumulative	
Function	Distortion	Speedup	Distortion	Speedup
pFL, inner	0.00%	1.252	0.00%	1.252
dist	0.00%	1.037	0.00%	1.708

swaptions (with bias adjustment)				
	Individual		Cumulative	
Function	Distortion	Speedup	Distortion	Speedup
Swaption_Blocking, outer	2.854%	1.983	2.854%	1.983
SimPath_Forward_Blocking	4.903%	1.014	8.593%	2.068
Swaption_Blocking, middle	1.111%	1.010	7.816%	2.129
Swaption_Blocking, inner	1.312%	1.014	7.990%	2.154

canneal				
	Individual		Cumulative	
Function	Distortion	Speedup	Distortion	Speedup
annealer_thread::run	7.467%	1.289	7.467%	1.289

blackscholes				
	Individual		Cumulative	
Function	Distortion	Speedup	Distortion	Speedup
main, outer	0.0%	1.898	0.0%	1.898

bodytrack				
	Individual		Cumulative	
Function	Distortion	Speedup	Distortion	Speedup
ParticleFilter::Update	0.050%	1.526	0.050%	1.526
ImageErrorInside, outer	0.038%	1.192	0.032%	1.809
ImageErrorInside, inner	0.028%	1.160	0.040%	1.957
GetObservation	0.371%	1.186	0.309%	2.291
BinaryImage, inner	0.023%	1.127	0.573%	2.409
BinaryImage, outer	0.036%	1.114	0.684%	2.449
MultiCameraProjectedBody, inner	0.146%	1.108	0.416%	2.752
MultiCameraProjectedBody, outer	0.062%	1.102	0.416%	2.935
ProjectedCylinder, outer	0.098%	1.036	0.968%	2.975
ProjectdCylinder, inner	0.056%	1.033	1.260%	3.036
OutputBMP	0.000%	1.017	1.260%	3.174
TrackingModel	0.064%	1.007	0.387%	3.214
FlexFilterRow, outer	0.000%	1.007	0.387%	3.247
FlexFilterRow, inner	0.078%	1.001	1.826%	3.284

Table 3: Loops selected for perforation in PARSEC benchmarks during training using 10% distortion bound.

behavior of the benchmark in the presence of core failures. For both figures, time, measured in heartbeats, is displayed on the x-axis, while performance is displayed on the left y-axis. Performance is measured using a sliding average over the last twenty heartbeats and it is shown for three scenarios. The first scenario is represented by the curve labeled "Baseline" and shows the performance of the system with no runtime and no core failures. The second scenario, represented by the curve labeled "No perforation" shows the performance of the system with core failures but without run-time support. The third scenario, labeled "SpeedGuard w/ perforation" shows the performance of the benchmark with SpeedGuard enabled in the presence of core failures. All performance is normalized to that of the baseline system. The points where core failures occur are the same for scenarios with and without perforation and these points are marked in the figures using a dashed vertical line. The right y-axis shows how the number of perforated loops varies dynamically as SpeedGuard responds to changes in performance.

7.1.1 Core failure in x264

Figure 16 shows the behavior of x264 in the presence of core failures. The "No perforations" curve shows that the core failures cause performance to fall to about 65% that of the baseline system. In contrast, SpeedGuard is able to respond to the core failures by dynamically perforating loops. As shown in the chart, when the core failures occur SpeedGuard begins perforating loops until performance returns to the desired value. By heartbeat 260, perforation has caused the application to exceed the maximal desired heart rate, so SpeedGuard reduces perforation. From that point on, SpeedGuard alternates between perforating three or four loops in an attempt to keep performance as close as possible to the desired value. By the end of the sequence performance is within 3% of the baseline system.

SpeedGuard is able to maintain the performance of x264 without resorting to video-specific fault tolerance methods like skipping frames. While common, the technique of dropping frames can have a large detrimental effect on user experience. For this example, the system with core failures would have to drop one out of every three frames, which would reduce a system with a frame rate of 25 frames per second to 16 frames per second. This drop in frame rate would be noticeable to the viewer as a stutter in the video. Furthermore, enabling an encoder to drop frames requires additional work for the programmer beyond the development of the encoder itself.

SpeedGuard is able to adjust to meet the performance goal with no measurable change to the quality of the video (measured in PSNR)[3]. SpeedGuard does increase the bitrate of the encoded video by 6%. This increase in bitrate could easily be tolerated by allocating a small amount of additional bandwidth or disk space. SpeedGuard is able to provide this service for the user with no additional burden other than inserting calls to the Heartbeat API.

7.1.2 Core Failure in bodytrack

Figure 16 shows the behavior of bodytrack in the presence of core failures. The "No perforation" curve shows that the core failures cause performance to fall to about 82% of the baseline system. As for x264, SpeedGuard is able to respond to the core failures by dynamically perforating loops. When the core failures occur, SpeedGuard begins perforating loops until performance returns to the desired value around heartbeat 90. For bodytrack, SpeedGuard consistently alternates between using three and four perforated loops to keep the average performance within the bounds. From heartbeat 90 until the end of the sequence the average performance is only 7% greater than that of the baseline system and well within the specified bounds.

We view the SpeedGuard system as "pushing" bodytrack along in this instance. When performance dips below a certain level SpeedGuard gives the application a push and it momentarily exceeds its goals, so the system backs off and stops pushing. Without the push, however, performance falls back below an acceptable level and the process repeats. Even though the instantaneous performance (measured using a 20 heartbeat sliding window) is not in the desired range, the average performance is, and these pushes allow bodytrack to keep up with its goals. In a real-time system, the periods where bodytrack runs below desired speed correspond to the application falling behind while data accumulates in a buffer. The momentary burst in speed (from the push) allows bodytrack to catch up and clear this buffer to keep it from overflowing.

SpeedGuard achieves this performance with less than 2% increase in distortion. The bodytrack benchmark outputs images that illustrate the track, and Figure 14 shows the output of the unmodified program while Figure 15 shows the output of the system which uses SpeedGuard to maintain performance in the face of core failure. The two images are almost identical, except that SpeedGuard has lost track of the location of the person's left forearm. Such a small error would likely not effect the performance of a robot or other autonomous system designed to interact with the human.

7.2 Frequency Scaling

In the second experiment, SpeedGuard responds to a dynamic change in core frequency. In this case, the benchmarks are run on an Intel Core 2 Duo T9400. The operating frequency of the chip can be adjusted dynamically using the cpufrequtils infrastructure of Linux [1]. Approximately one quarter of the way through the computation the chip frequency is reduced from 2.53 GHz to 1.6 GHz. Then, approximately three quarters of the way through the computation, the frequency is reset to its original value. The target performance for both benchmarks is established using the Heartbeat API. Both applications request a minimal heart rate that is no less than 90% and no more than 115% of what they achieve in a system with no frequency scaling.

Figures 18 and 19 show the dynamic behavior of the benchmarks in reaction to a dynamic change in processor frequency. As in the core failure experiment, time is displayed on the x-axis, while performance is displayed on the left y-axis. Performance is again measured using a sliding average over the last twenty heartbeats and it is shown for the baseline scenario (with no frequency change), a scenario with sudden frequency change but no perforation, and the scenario with a frequency change, but with SpeedGuard enabled. The points where frequency changes are labeled in the figures using dashed vertical lines. The right y-axis shows how the number of perforated loops varies dynamically.

We note that the frequency scaling experiment demonstrates not only the ability to use code perforation to respond to faults, but also the ability to trade accuracy for power savings. Power in a microprocessor is proportional to cv^2f, where c is capacitance, v is voltage and f is frequency. In these experiments, processor frequency is reduced by 36% and SpeedGuard maintains performance. This allows us to effectively reduce power by 36% in exchange for some accuracy loss; however, if the processor operates at lower frequency, we could also lower the voltage for further reductions in power. In fact, the processor used in this study does lower voltage when frequency is lowered. When voltage is also reduced from 1.3V to 1.1V, the total power savings is over 2.2×.

[3] Output videos are available at http://www.youtube.com/view_play_list?p=0347D028F143EA93.

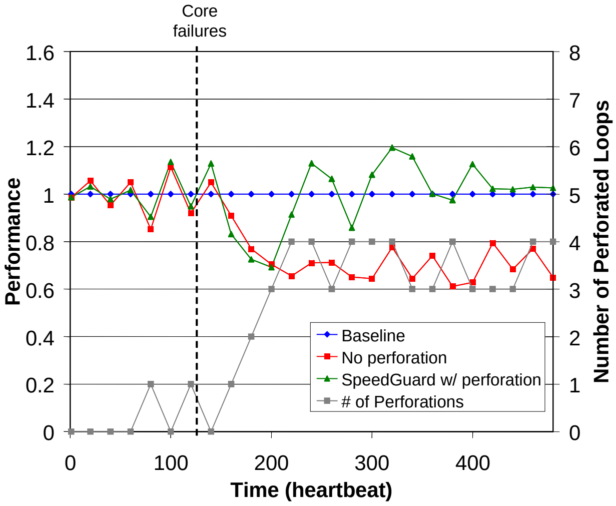

Figure 16: Dynamic behavior of SpeedGuard for x264 benchmark in the presence of three core failures.

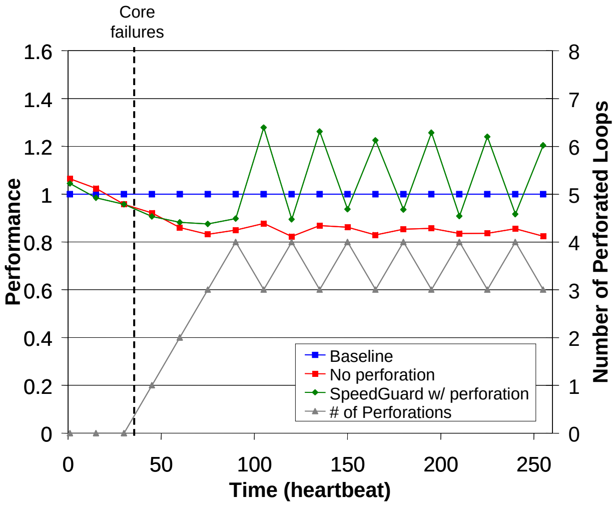

Figure 17: Dynamic behavior of SpeedGuard for bodytrack benchmark in the presence of three core failures.

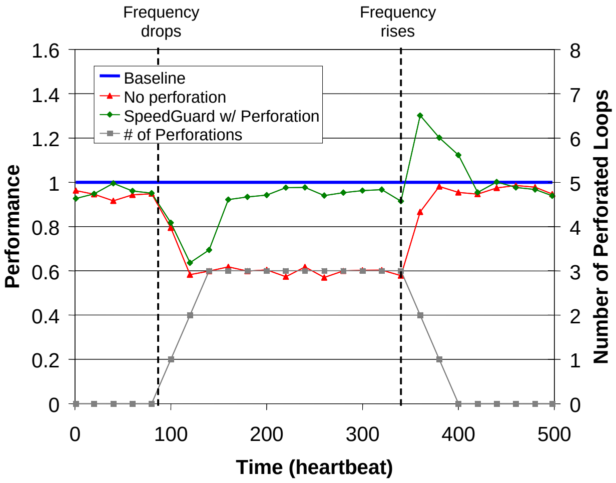

Figure 18: Dynamic behavior of SpeedGuard for x264 benchmark in the presence of frequency scaling.

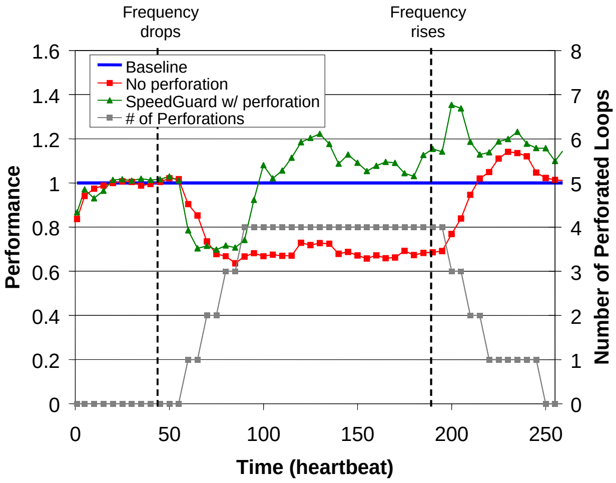

Figure 19: Dynamic behavior of SpeedGuard for bodytrack benchmark in the presence of frequency scaling.

7.2.1 *Frequency scaling in x264*

Figure 18 shows the behavior of x264 during a sudden change in processor frequency. The "No perforation" curve shows that this sudden decrease in computing power causes performance to fall to about 65% that of the baseline system by heartbeat 120. Performance remains at this level until the frequency increases and then it returns to the expected level. In contrast, SpeedGuard is able to respond to the frequency change by dynamically perforating loops. As shown in the chart, when the frequency change occurs SpeedGuard perforates three loops and the performance returns to the desired value by heartbeat 160. When the frequency is raised to its original value, SpeedGuard quickly reduces the number of perforations so that by heartbeat 400, the program is again running unperforated code.

As in the core failure experiment, SpeedGuard dynamically adjusts perforation level to meet the application's performance goals with no measurable change to the quality of the video. In this case, SpeedGuard only increases the bitrate of the encoded video by 2%. Again, for a real-time encoder, this small and temporary increase to bitrate might be preferable to a few seconds of choppy video.

7.2.2 *Frequency scaling in bodytrack*

Figure 19 shows the behavior of bodytrack during a sudden change in processor frequency. The "No perforation" curve shows that the sudden loss of processing power causes performance to fall to about 60% that of the baseline system by heartbeat 85. Performance remains at this level until the frequency increase allows it to return to the desired level. In contrast, SpeedGuard is able to respond to the frequency change by dynamically perforating loops. As shown in the chart, when the frequency change occurs SpeedGuard perforates three loops and the performance returns to the desired value by heartbeat 100. When the frequency increases, SpeedGuard quickly reduces the number of perforations so that by heartbeat 250 bodytrack is again running unperforated code.

As in the core failure experiment, SpeedGuard is able to meet bodytrack's performance goals with only a small degradation of its ability to track the body through the scene. Perforation causes some inaccuracies in the position of the subject's forearm, but we have already identified several applications where this might be preferable to failing to return any result promptly.

7.3 Summary of Fault Tolerance Experiments

These experiments demonstrate several important features of the SpeedGuard system. First, they demonstrate how the performance gains found by SpeedPress can be converted into robustness in the face of errors. Second, this study highlights SpeedGuard's flexibility to respond to different types of faults without changing the application code or even the compiled binary. SpeedGuard achieves this flexibility by detecting performance changes instead of particular faults, meaning that the system can respond to any environmental change that causes a variation in performance. Third, these results are achieved with no application specific fault tolerance code and no additional burden on the user apart from the required calls to the heartbeat interface.

8. RELATED WORK

Trading accuracy of computation for other optimizations is a well-known technique. It has been shown that one can trade off accuracy for performance [22], robustness [22], energy consumption [10, 31, 22] and fault tolerance [10, 31, 22]. We note that developers have, for years, manually navigated performance/accuracy tradeoffs. For example, the Search benchmark in the Jade benchmark set [9, 25] was manually optimized to use more efficient, less accurate, but still acceptably accurate alternate implementations of mathematical functions such as `sqrt`, `sin`, and `cos`.

8.1 Software Techniques

Rinard [22, 23] presents a technique for automatically deriving empirical probabilistic distortion and timing models that characterize the accuracy/performance tradeoff space of a given application. To the best of our knowledge, this research is the first to propose and infer such models. Given a program that executes a set of tasks, these models characterize the effect of skipping task executions on the performance and execution time. The results show that skipping tasks can often significantly reduce the execution time (because the program performs less computational work) while producing acceptable changes in the output. Note that because the execution time reductions correspond directly to reductions in the amount of computational resources required to perform the computation, execution time reductions correspond directly to energy reductions. The research evaluates the use of these models to tolerate task failures (or failures of the underlying computational platform executing the task), purposefully discard tasks to reduce the execution time and/or the energy consumption, and eliminate idle time at barriers terminating parallel phases, all while keeping the resulting output distortion within acceptable bounds. Many of the tasks in the reported benchmark set execute subsets of loop iterations. Discarding such tasks has essentially the same effect as loop perforation. This research was the original inspiration for the research presented in this paper.

In comparison, the primary advantage of loop perforation is its ability to operate on applications written in standard languages without the need for the developer to identify task boundaries. This paper also demonstrates how to use loop perforation in combination with dynamic performance monitoring techniques and fine-grain control over skipped computation to enable applications with real-time performance goals to adapt to events such as core failures and clock frequency changes.

Researchers have also developed systems that allow developers to provide multiple different implementations for a given piece of functionality, with different implementations occupying different points in the performance/accuracy trade off space. Petabricks [3] is a new parallel language and compiler that developers can use to provide multiple alternate implementations of a given piece of functionality, each with potentially different accuracy/performance characteristics. Given these alternatives, the compiler and runtime system can dynamically select and create an optimized hybrid algorithm that is tailored for solving a particular problem given input characteristics (e.g. size) and environmental constraints such as the level of available parallelism (e.g. number of cores).

Green [5] also provides constructs that developers can use to provide alternate implementations for given pieces of functionality, with the alternate implementations typically offering better performance or energy consumption characteristics but less accuracy. One of the supported alternate implementations is loop approximation, or loops that exit early, effectively skipping the remaining contiguous set of loop iterations, an optimization similar to our truncation perforation. The Green framework requires programmers to annotate their code using extensions to C and C++ and to provide additional functions needed by the Phoenix compiler framework.

In contrast to these techniques, this paper presents an automated solution that works directly on original unmodified applications. There is no need for the developer to provide multiple implementations of any piece of functionality, no need for the developer to understand the characteristics of the application and its implementation to identify which approximations are likely to provide reasonable performance gains with acceptable accuracy, and no need to understand or modify the source code of the application at all. In contrast, our implemented system automatically generates a large performance/accuracy tradeoff space and automatically searches that space to find points that simultaneously meet both performance and accuracy goals.

We also present the SpeedGuard framework, which, combined with the Application Heartbeats framework, enables our system to automatically control the amount of applied perforation to enable effective automatic responses to a wide variety of environmental changes that affect performance and energy consumption, including dynamic changes in the clock frequency.

8.2 Hardware Techniques

Considerable research has gone into exploring how to trade off accuracy for reduced energy consumption. George et al [15] show how circuit level errors can be ignored if the application level impact is low and if such a tradeoff leads to considerable gains in energy efficiency. Chakrapani [10] extends that work with a more detailed theoretical model of the tradeoff between energy consumption and error induced by propagation delay in circuits that implement arithmetic operations that can be exploited to gain energy savings. Deviation-tolerant computation [31] present an analysis of how one can trade off correctness, in the presence of hardware faults, for performance or energy efficiency. Their method for trading accuracy for performance models how altering an external system feature (e.g. circuit noise margins) may result in a performance increase at the cost of higher error probability.

These techniques are complementary to our work as they operate at a different scope. They show to trade performance for accuracy at a bit, or sub-word level (e.g. simple ripple-carry adders) while our approach deals with performance/accuracy trade-offs at a the level of loops and function calls. Additionally, our method for trading off accuracy relies on skipping the execution of code rather than tolerating errors at the hardware level. Finally, we have implemented our techniques in a real system including both the SpeedPress compiler and SpeedGuard runtime system.

8.3 Feedback-Driven Optimization

One way to view code perforation is an optimizing compiler that examines the accuracy-performance space to drive program optimizations. Optimizing programs based on run time profiling is a well studied area [30]. Feedback-directed optimization (FDO) is general term used to describe optimization techniques that alter a program's execution based on information gathered at run time [30]. FDO techniques range from static to dynamic. At the static end of the spectrum, run time profiling information is used to produce new, optimized, binaries (recompilation) [11, 29, 17]. Fully dynamic hardware [26, 20] and software techniques[6, 12, 4] are at the other end of the spectrum. All of these techniques operate under the constraint that the transformed program must produce the identical output as the original program. Code perforation, in contrast, is designed to produce programs whose outputs may differ within acceptable distortion bounds. This additional freedom enables code perforation to, in general, deliver larger performance increases.

8.4 Fault Tolerance

Recently, several techniques that trade off correctness for system availability have been proposed. Examples include failure-oblivious computing [24], error virtualization [27, 28], DieHard [7], acceptability-oriented computing [21] and data-structure repair [13]. These techniques exploit the concept of *software elasticity:* the ability of regular code to recover from certain types of failures when low-level faults are masked by the operating system (OS) or by appropriate instrumentation. When applied to fault tolerance, these techniques are useful for recovering from failures that cause memory errors but cannot respond to failures that cause system performance degradation. Our approach also harnesses software elasticity, but instead responds to errors affecting system performance rather than memory.

9. CONCLUSIONS

We have presented and evaluated a technique, code perforation, for automatically enabling applications to increase their performance on demand while keeping any resulting output distortion within acceptable bounds. Our experimental results show that this technique can enable applications to deliver acceptable results to users more quickly. It can also enable applications to automatically adapt to meet real-time performance goals in the face of disruptive events such as core failures, clock speed changes, and increased loads.

We acknowledge that code perforation is not appropriate for all possible applications. But within its target class of applications, our results show that it can automatically deliver an important capability that dramatically increases the ability of systems to deal successfully with complex and dynamically changing combinations of performance demands, failures, and accuracy requirements.

10. REFERENCES

[1] cpufrequtils. `http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html`.

[2] E. Amigo, J. Gonzalo, and J. Artiles. A comparison of extrinsic clustering evaluation metrics based on formal constraints. In *Information Retrieval Journal*. Springer Netherlands, jul 2008.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeno JVM. In *OOPSLA '00*, pages 47–65. ACM New York, NY, USA, 2000.

[5] W. Baek and T. Chilimbi. Green: A system for supporting energy-conscious programming using principled approximation. Number TR-2009-089, Aug. 2009.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12. ACM New York, NY, USA, 2000.

[7] E. Berger and B. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI*, June 2006.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT-2008: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

[9] R. Browning, T. Li, B. Chui, J. Ye, R. Pease, Z. Czyzewski, and D. Joy. Low-energy electron/atom elastic scattering cross sections from 0.1-30 keV. *Scanning*, 17(4), 1995.

[10] L. Chakrapani, K. Muntimadugu, A. Lingamneni, J. George, and K. Palem. Highly energy and performance efficient embedded computing through approximately correct arithmetic: A mathematical foundation and preliminary experimental validation. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008.

[11] R. Cohn and P. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*, November 1999.

[12] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization*, pages 15–24, 2003.

[13] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05*, pages 176–185, New York, NY, USA, 2005. ACM.

[14] B. Furht, J. Greenberg, and R. Westwater. *Motion Estimation Algorithms for Video Compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[15] J. George, B. Marr, B. Akgul, and K. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 158–168. ACM New York, NY, USA, 2006.

[16] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal. Application Heartbeats for Software Performance and Health. Technical Report TR-2009-035, Computer Science and Artificial Intelligence Laboratory, MIT, Aug. 2009.

[17] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, et al. The superblock: an Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1):229–248, 1993.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] S. Muchnick. *Advanced Compiler Design and*

Implementation. Morgan Kaufman Publishers, 1997.

[20] D. Patel and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *31st Annual Acm/IEEE International Symposium on Microarchitecture*, page 173. IEEE Computer Society, 1998.

[21] M. Rinard. Acceptability-oriented computing. In *OOPSLA '03*, Oct. 2003.

[22] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM New York, NY, USA, 2006.

[23] M. Rinard. Using early phase termination to eliminate load imbalancess at barrier synchronization points. Oct. 2007.

[24] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, December 2004.

[25] M. Rinard, D. Scales, and M. Lam. Jade: A high-level, machine-independent language for parallel programming. *parallel computing*, 29, 1993.

[26] E. Rotenberg. *Trace processors: Exploiting hierarchy and speculation*. PhD thesis, 1999.

[27] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS '09*, 2009.

[28] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX*, Apr. 2005.

[29] M. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the first SUIF compiler workshop*, pages 14–25, 1996.

[30] M. Smith. Overcoming the challenges to feedback-directed optimization (Keynote Talk). *ACM SIGPLAN Notices*, 35(7):1–11, 2000.

[31] P. Stanley-Marbell, D. Dolech, A. Eindhoven, and D. Marculescu. Deviation-Tolerant Computation in Concurrent Failure-Prone Hardware. 2008.

[32] x264. http://www.videolan.org/x264.html.