Patterns and Statistical Analysis for Understanding Reduced Resource Computing

Martin Rinard
Massachusetts Institute of Technology
rinard@mit.edu

Henry Hoffmann
Massachusetts Institute of Technology
hank@csail.mit.edu

Sasa Misailovic
Massachusetts Institute of Technology
misailo@csail.mit.edu

Stelios Sidiroglou
Massachusetts Institute of Technology
stelios@csail.mit.edu

Abstract

We present several general, broadly applicable mechanisms that enable computations to execute with reduced resources, typically at the cost of some loss in the accuracy of the result they produce. We identify several general computational patterns that interact well with these resource reduction mechanisms, present a concrete manifestation of these patterns in the form of simple model programs, perform simulation-based explorations of the quantitative consequences of applying these mechanisms to our model programs, and relate the model computations (and their interaction with the resource reduction mechanisms) to more complex benchmark applications drawn from a variety of fields.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Patterns; D.2.4 [*Software/Program Verification*]: Statistical Methods; D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Design, Measurement, Performance, Reliability, Verification

Keywords Reduced Resource Computing, Discarding Tasks, Loop Perforation, Cyclic Memory Allocation, Statistical Analysis

Onward! 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

1. Introduction

The amount of available resources is a central factor in the existence of virtually all living organisms. Mechanisms that adapt the operation of the organism to variations in resource availability occur widely throughout nature. For example, during prolonged starvation, the human body preserves muscle mass by shifting its fuel source from proteins to ketone bodies [3]. Peripheral vasoconstriction, which minimizes heat loss by limiting the flow of blood to the extremeties, is a standard response to hypothermia. The nasal turbinates in dehydrated camels extract moisture from exhaled respiratory air, thereby limiting water loss and enhancing the ability of the camel to survive in dessicated environments [31]. All of these mechanisms take the organism away from its preferred operating mode but enable the organism to degrade its operation gracefully to enhance its survival prospects in resource-poor environments.

The vast majority of computer programs, in contrast, execute with essentially no flexibility in the resources they consume. Standard programming language semantics entails the execution of every computation the program attempts to perform. If the memory allocator fails to return a valid reference to an allocated block of memory, the program typically fails immediately with a thrown exception, failed error check, or memory addressing error. This inability to adapt to changes in the underlying operating environment impairs the flexibility, robustness, and resilience of almost all currently deployed software systems.

Reduced resource computing encompasses a set of mechanisms that execute programs with only a subset of the resources (time and/or space) that the standard programming language semantics and execution environment provides. Specific reduced resource computing mechanisms include:

- **Discarding Tasks:** Parallel computations are often structured as a collection of tasks. Discarding tasks produces new computations that execute only a subset of the tasks in the original computation [23, 24].
- **Loop Perforation:** Loop perforation transforms loops to execute only a subset of the iterations in the original computation [16, 20]. Different loop perforation strategies include modulo perforation (which discards or executes every nth iteration for some fixed n), truncation perforation (which discards either an initial or final block of iterations), and random perforation (which discards randomly selected iterations).
- **Cyclic Memory Allocation:** Cyclic memory allocation allocates a fixed-size buffer for a given dynamic allocation site [21]. At each allocation, it returns the next element in the buffer, wrapping back around to the first element when it reaches the end of the buffer. If the number of live objects allocated at the site is larger than the number of elements in the buffer, cyclic memory allocation produces new computations that execute with only a subset of the memory required to execute the original computation.

1.1 Resource Reduction in Practice

Unsurprisingly, these mechanisms almost always change the output that the program produces. So they are appropriate only for computations that have some flexibility in the output they produce. Examples of such computations include many numerical and scientific computations, sensory applications (typically video and/or audio) that involve lossy encoding and decoding, many machine learning, statistical inference, and finance computations, and information retrieval computations. The relevant question is whether these kinds of computations are still able to deliver acceptable output after resource reduction.

Interestingly enough, our empirical results show that many of these computations contain components that can successfully tolerate the above resource reduction mechanisms — the computation still produces acceptably accurate outputs after the application of these mechanisms to these components. And these resource reduction mechanisms can often endow computations with a range of capabilities that are typically otherwise available only through the manual development of new algorithms. Specifically, discarding tasks has been shown to enable computations to tolerate task failures without retry [23], to produce accuracy and performance models that make it possible to purposefully and productively navigate induced accuracy versus performance trade-off spaces (for example, maximizing accuracy subject to performance constraints or maximizing peformance subject to accuracy constraints) [23], and to eliminate barrier idling at the end of parallel loops [24]. Cyclic memory allocation has been shown to eliminate otherwise potentially fatal memory leaks [21]. Loop perforation has been shown to reduce the overall execution time of the computation and to enable techniques that dynamically control the computation to meet real-time deadlines in the face of clock rate changes and processor failures [16].

A key to the successful application of these mechanisms in practice is identifying the components that can successfully tolerate resource reduction, then applying resource reduction only to those components. This empirical fact leads to usage scenarios in which the resource reduction mechanisms generate a search space of programs close to the original programs. An automated (or semiautomated) search of this space finds the components that can tolerate resource reduction, with resource reduction confined to those components when the computation executes. The remaining components execute with the full set of resources with which they were originally designed to operate. The resulting effect is conceptually similar to the mechanisms that biological organisms use to deal with reduced resources, which direct the delivery of scarce resources to those critical functions most necessary for survival.

1.2 Inherent Redundancy

The success of reduced resource computing shows that many computations, like biological organisms, have inherent sources of redundancy that enable them to operate successfully in the face of reduced resources. Note, however, that *these sources of redundancy were not explicitly engineered into the computation — they emerge as an unintended consequence of the way the computation was formulated*. In this paper we analyze various sources of redundancy that enable these computations to tolerate resource reduction.

The result of this analysis is several general computational patterns that interact in very reasonable ways with the different resource reduction mechanisms. Viewing our computations through the prism of these patterns helped us understand the behavior we were observing; we anticipate that recognizing these patterns in other computations will facilitate the prediction of how these other computations will react to resource reduction.

In the future, trends such as the increasing importance of energy consumption, the need to dynamically adapt to computing platforms with fluctuating performance, load, and power characteristics, and the move to more distributed, less reliable computing platforms will increase the need for computations that can execute successfully across platforms with a range of (potentially fluctuating) available resources. Initially, we expect developers to let automated techniques find and exploit patterns in existing applications that interact well with resource reduction. They may then move on to deploying such patterns into existing applications to enhance their ability to function effectively in a range of environments. Ultimately, we expect developers to engineer software systems from the start around patterns that interact well with resource reduction in much the same way that developers now work with more traditional design patterns [12] in all phases of the engineering process.

1.3 Contributions

This paper makes the following contributions:

- **Computational Patterns:** It identifies computational patterns that interact well with resource reduction mechanisms such as discarding tasks, perforating loops, and dynamically allocating memory out of fixed-size buffers. Understanding these patterns can help developers develop a conceptual framework that they can use to reason about the interaction of their applications with various resource reduction mechanisms.
- **Model Computations:** It presents concrete manifestations of the general patterns in the form of simple model computations. These model computations are designed to capture the essential properties of more complex real-world applications that enable these applications to operate successfully in the presence of resource reduction mechanisms.

  In this way, the model computations can give developers simple, concrete examples that can help them think productively about the structure of their applications (both existing and envisioned) and how that structure can affect the way their applications will respond to the application of different resource reduction mechanisms.

  The model computations can also serve as the foundation for static analyses that recognize computations that interact well with resource reduction mechanisms. Such analyses could produce statistical models that precisely characterize the effect of resource reduction mechanisms on the application at hand, thereby making it possible to automatically apply resource reduction mechanisms to obtain applications with known statistical accuracy properties in the presence of resource reduction.
- **Simulations:** It presents the results of simulations that use the model computations to quantitatively explore the impact of resource reduction on the accuracy of the results that the computations produce. The simulation results can help developers to better estimate and/or analyze the likely quantitative accuracy consequences of applying resource reduction mechanisms to their own applications.
- **Relationship to Applications:** It relates the structure of the model computations and the simulation accuracy results back to characteristics of specific benchmark applications. Understanding these relationships can help developers better understand the relationships between the model computations and their own applications.
- **A New Model of Computation:** Standard models of computation are based on formal logic [11, 14]. In these models, the computation is rigidly fixed by the application source code, with formulas in discrete formal logic characterizing the relationship between the input and output. This paper, in contrast, promotes a new and fundamentally different model in which the computation is flexible and dynamic, able to adapt to varying amounts of resources, and characterized by (conceptually) continuous statistical relationships between the input, output, and amount of resources that the computation consumes.

  Of course, almost every program has some hard logical correctness requirements — even a video encoder, for example, must produce a correctly formatted video file (even though it has wide latitude in the accuracy of the encoded video in the file). We therefore anticipate the development of new hybrid analysis approaches which verify appropriate hard logical correctness properties using standard program analysis techniques but use new statistical techniques to analyze those parts of the computation whose results can (potentially nondeterministically) vary as long as they stay within acceptable statistical accuracy bounds.

2. The Mean Pattern

Consider the following computations:

- **Search:** The Search computation [7] from the Jade benchmark suite [28] simulates the interaction of electron beams with solids. It uses a Monte-Carlo simulation to track the paths of electrons, with some electrons emerging back out of the solid and some remaining trapped inside. The program simulates the interaction for a variety of solids. It produces as output the proportion of electrons that emerge out of each solid. Each parallel task simulates some of the electron/solid interaction pairs.
- **String:** The String computation [13] from the Jade benchmark suite [25] uses seismic travel-time inversion to compute a discrete velocity model of the geology between two oil wells. It computes the travel time of rays traced through the geology model, then backprojects the difference between the ray tracing times and the experimentally observed propagation times back through the model to update the individual elements in the velocity model through which the ray passed. Each parallel task traces some of the rays.

The core computations in both Search and String generate sets of numbers, then compute the mean of each set. In Search, each number is either one (if the electron emerges from the solid) or zero (if the electron is trapped within the solid). There is a single set of ones and zeros for each solid; the output is the mean of the set. In String there is one set of numbers for each element of the discrete velocity model. Each number is a backprojected difference from one ray that traversed the element during its path through the geology model. String combines the numbers in each set by computing their mean. It then uses these numbers to update the corresponding elements of the velocity model.

The resource reduction mechanism for both computations, discarding tasks, has the effect of eliminating some

of the numbers from the sets. It is possible to derive empirical linear regression models that characterize the effect of this resource reduction mechanism on the output that these two computations produce [23]. These models show that discarding tasks has a very small impact on the output that the computation produces. Specifically, the models indicate that discarding one quarter of the tasks changes the Search output by less than 3% and the String output by less than 1%; discarding half of the tasks (which essentially halves the running time) changes the Search output by less than 6% and the String output by less than 2%.

2.1 The Model Computation

Our model computation for these two computations simply computes the mean of a set of numbers:

```
for (i = 0; i < n; i++) {
  sum += numbers[i];
  num++;
}
mean = sum/num;
```

The resource reduction mechanism for this model computation executes only a subset of the loop iterations to compute the mean of a subset of the original set of numbers. The specific mechanism we evaluate (loop perforation) simply discards every other number when it computes the sum by executing only every other iteration in the model computation:

```
for (i = 0; i < n; i += 2) {
  sum += numbers[i];
  num++;
}
mean = sum/num;
```

We evaluate the effect of this resource reduction mechanism via simulation. Our first simulation works with floating point numbers selected from a continuous probability distribution. Each trial in the simulation starts by filling the array `numbers` with the values of `n` independent pseudorandom variables selected from the uniform distribution over the interval $[0, 1]$. It then computes the difference between the computed `mean` values with and without resource reduction — i.e., the difference between the mean of all `n` values in the `numbers` array and the mean of every other value in the `numbers` array. For each even value of `n` between 10 and 100, we perform 1,000,000 such trials.

Our second simulation works with integers selected from a discrete probability distribution. Each trial fills the array `numbers` with the values of `n` psuedorandom variables selected from the uniform discrete distribution on the set $\{0, 1\}$. We then perform the same simulation as detailed above for the continuous distribution.

Figure 1 presents the results of these trials. This figure presents four graphs. The x axes for all graphs range over the sizes (values of `n`) of the sets in the trials. The upper left graph plots the mean of the differences between the results that the standard and resource reduced computations produce (each point corresponds to the mean of the observed differences for the 1,000,000 trials for sets of the corresponding size) for the continuous distribution. The upper right graph plots the variances of the differences for the continuous distribution. The lower left graph plots the mean of the differences for the discrete distribution; the lower right graph plots the corresponding variances.

These numbers show that our model computation exhibits good accuracy properties in the presence of resource reduction. Specifically, for all but the smallest sets of numbers, resource reductions of a factor of two cause (in most cases substantially) less than a 10% change in the result that the computation produces. We attribute this robustness in the face of resource reduction to a diffuse form of partial redundancy that arises from the interaction of the computation with the data on which it operates. Because the numbers in the reduced resource computation are a subset of the complete set of numbers and because the numbers are all drawn from the same probability distribution, the two mean values are highly correlated (with the correlation increasing as the size of the sets increases).

We note that the graphs show that the accuracy of the resource reduced computation depends on the size of the set of numbers, with larger sets producing smaller differences and variances than larger sets of numbers. This phenomenon is consistent with our redundancy-based perspective, since smaller sets of numbers provide our model computation with less redundancy than larger sets of numbers.

The discrete distribution has higher mean differences and variances than the continuous distribution, which we attribute to the concentration of the weight of the discrete probability distribution at the extremes. We also note that all of our simulation numbers are for resource reductions of a factor of two, which is much larger than necessary for many anticipated scenarios (for example, scenarios directed towards tolerating failures).

2.2 Relationship to Search and String

The model computation provides insight into why the applications tolerate the task discarding resource reduction mechanism with little loss in accuracy. While discarding tasks may, in principle, cause arbitrary deviations from the standard behavior of the application, the underlying computational patterns in these applications (although obscured by the specific details of the realization of the patterns in each application) interact well with the resource reduction mechanism (discarding tasks). The final effect is that the resource reduction mechanism introduces some noise into the computed values, but has no other systematic effect on the computation. And in fact, the results from our model computation show that it is possible to discard half the tasks in the computation with (depending on the size of the set of numbers) single digit percentage accuracy losses. This result

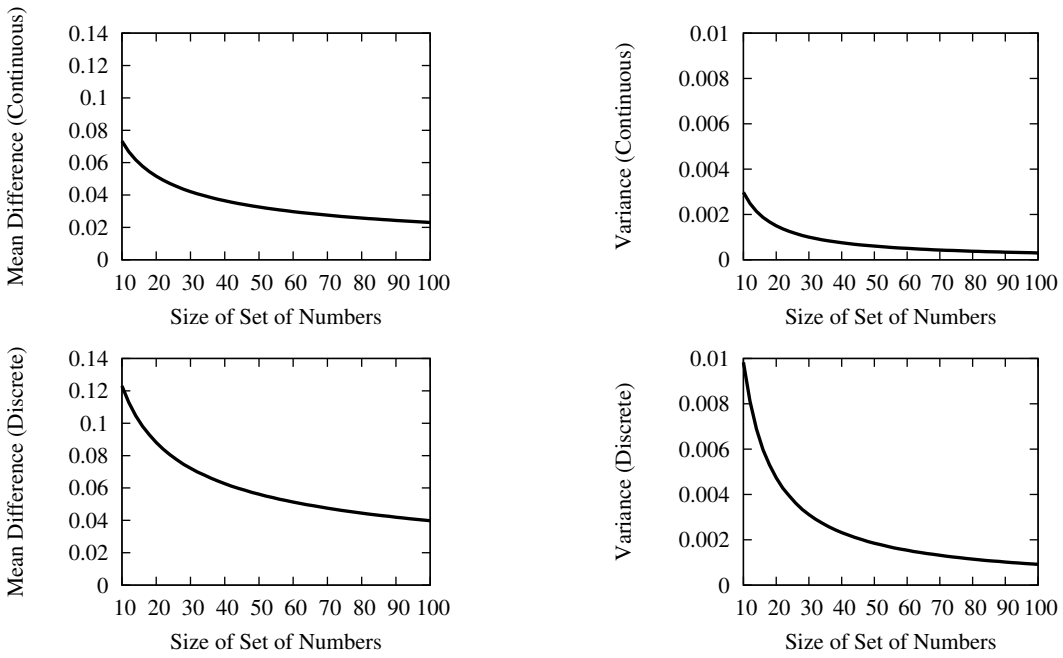

Figure 1. Means and Variances of Differences Between Standard and Resource Reduced Mean Computations

from our model computation is consistent with the results from both the Search and String applications. The larger mean differences and variances for the discrete distribution show that the Search application (in which each number is either 0 or 1) needs to work with larger sets than the String application (in which each number is a floating point number) to obtain similar accuracies under similar resource reduction factors.

3. The Sum Pattern

Consider the following computations:

- **Water:** Water evaluates forces and potentials in a system of molecules in the liquid state [25]. Although the structure is somewhat obscured by the application models the interactions between the water molecules, the core computations in Water boil down to computing, then taking the sum of, sets of numbers. For example, a key intermolecular force calculation computes, for each water molecule, the sum of the forces acting on that water molecule from all of the other water molecules. Water is coded as a Jade program [25], with each task computing, then taking the sum of, a subset of the corresponding set of numbers.
- **Swaptions:** Swaptions uses a Monte-Carlo simulation to solve a partial differential equation to price a portfolio of swaptions [5]. The core computation takes the sum of the results from the individual simulations. The application computes the final result by dividing the sum by the number of simulations.

The resource reduction mechanism for Water is discarding tasks [23, 24]; the resource reduction mechanism for Swaptions is loop perforation [16, 20]. In both cases the effect is a reduction in the result proportional to the number of discarded tasks or loop iterations. Unlike the Search and String computations discussed in Section 2, for Water and Swaptions discarding many tasks or loop iterations can therefore induce a large change in the overall result that the computation produces.

3.1 The Model Computation

The model computation for Water and Swaptions computes the sum of a set of psuedorandom numbers.

```
for (i = 0; i < n; i++) sum += numbers[i];
```

As for the mean model computation, the resource reduction mechanism is to discard every other iteration of the loop:

```
for (i = 0; i < n; i += 2) sum += numbers[i];
```

The effect is to divide the result (the value of the `sum` variable) by approximately a factor of two. It is possible, however, to use extrapolation to restore the accuracy of the computation [16, 23] - simply multiply the final result by two, or, more generally, the number of tasks or loop iterations in the original computation divided by the number of tasks or loop iterations in the resource reduced computation. Note that the former number (the number of tasks or loop iterations in the original computation) is typically available in the resource reduced computation as a loop bound (for our model computation, `n`) or some other number used to control the generation of the computation. After extrapolation, the accuracy

picture is essentially similar to the accuracy picture for the computations with the mean pattern (see Section 2) — the extrapolated sum is simply the mean multiplied by the appropriate ratio.

3.2 Relationship to Water and Swaptions

Water and Swaptions tolerate resource reduction for essentially the same reason that Search and String tolerate resource reduction — all of these computations combine a set or sets of partially redundant numbers together into a final result or results, with addition as the basic combination mechanism. In all of the applications the resource reduction mechanism has the effect of eliminating some of the numbers from the combination, with the redundancy present in the numbers enabling the computations to tolerate the elimination with small accuracy losses.

4. The Minimum Sum Pattern

x264 is an H.264 video encoder from the Parsec benchmark suite [5]. It encodes a sequence of frames, with each frame encoded as a collection of blocks (rectangular regions of the frame). One way to encode a block is to reference a block in a previously encoded frame, then specify the contents of the block as a set of differences relative to the referenced block. The more similar the blocks are, the more efficient the encoding. The search for a similar block in a previously encoded frame is called motion estimation [18]. x264 spends the majority of its time in motion estimation.

The following code fragment implements a comparison between an encoded block and a reference block. It computes the sum of a set of numbers designed to characterize the visual difference between the two blocks (the variable `value` holds the sum). This code block executes for a variety of potential reference blocks, with the computation choosing to use the block with the smallest sum as the final reference block for the relative encoding (if that encoding is more compact than the standard discrete cosine transformation encoding).

```
for (i = 0; i < h; i += 4) {
  for (j = 0; j < w; j += 4) {
    element_wise_subtract(temp, cur,
                          ref, cs, rs);
    hadamard_transform(temp, 4);
    value += sum_abs_matrix(temp, 4);
  }
  cur += 4*cs; ref += 4*rs;
}
return value;
```

For this application, we consider the loop perforation resource reduction mechanism [16, 20]. In this example loop perforation coarsens the comparison by changing the loop increments from 4 to 8. Note that because of the loop nesting there is a multiplicative effect on the running time — the loop nest runs four, not just two, times faster.

```
for (i = 0; i < h; i += 8) {
  for (j = 0; j < w; j += 8) {
    element_wise_subtract(temp, cur,
                          ref, cs, rs);
    hadamard_transform(temp, 4);
    value += sum_abs_matrix(temp, 4);
  }
  cur += 4*cs; ref += 4*rs;
}
return value;
```

We measure the quality of the encoder using two measures: the peak signal to noise ratio and the bitrate of the encoded video. Perforating the outer loop alone produces an encoder that runs 1.46 times faster than the original. The peak signal to noise ratio decreases by 0.64% and the bitrate increases by 6.68%. Perforating the inner loop alone makes the encoder run 1.45 times faster, the peak signal to noise ratio decreases by 0.46%, and the bitrate increases by 8.85%. Finally, perforating both loops makes the encoder run 1.67 times faster, the peak signal to noise ratio decreases by 0.87%, and the bitrate increases by 18.47%. All versions of the encoded videos are visually indistinguishable.

4.1 The Model Computation

The model computation starts with a collection of `m` sets of `n` numbers. It finds the set with the smallest sum.

```
min = DBL_MAX;
index = 0;
for (i = 0; i < m; i++) {
  sum = 0;
  for (j = 0; j < n; j++) sum += numbers[i][j];
  if (min < sum) {
    min = sum;
    index = i;
  }
}
```

We consider two resource reduction mechanisms for this computation. The first transforms the inner loop (over `j`) so that it performs only a subset of the iterations. The specific transformation is loop perforation, which executes every other iteration instead of every iteration. The resource reduced computation increments `i` by 2 after each loop iteration instead of 1.

The resulting computation takes the sum of only a subset of the numbers in each set (in this case the sum of every other number) rather than the sum of all of the numbers. The computation may therefore compute the index of a set in which the sum of the remaining unconsidered numbers in the set may add up to a value large enough to make the sum of all of the numbers of the set larger than the sum of the set with the minimum sum. Conceptually, this transformation corresponds to the application of loop perforation to the x264 application as described above.

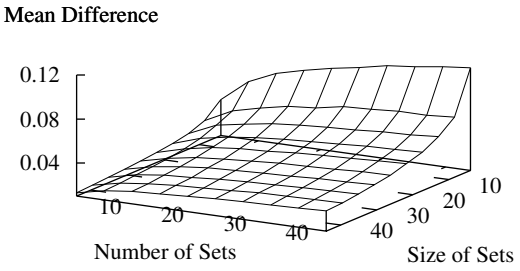

Figure 2. Mean Differences With Resource Reduction on Each Set

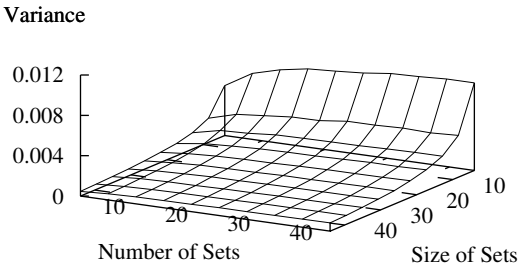

Figure 3. Variance in Differences With Resource Reduction on Each Set

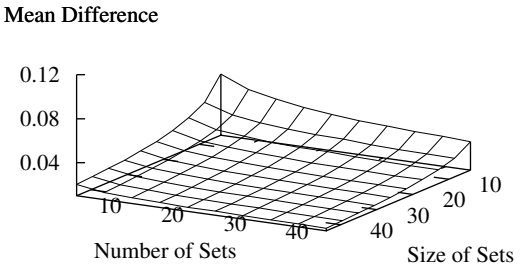

Figure 4. Mean Differences With Resource Reduction on Number of Sets

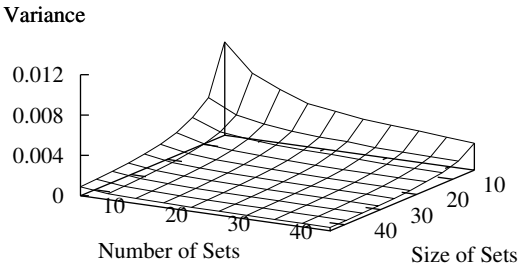

Figure 5. Variance in Differences With Resource Reduction on Number of Sets

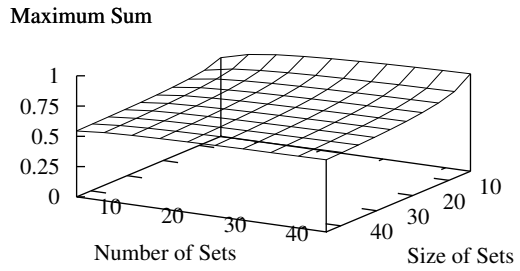

Figure 6. Mean Maximum Sums

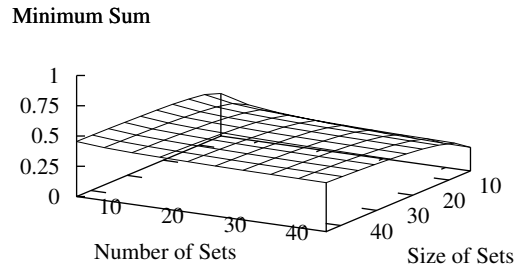

Figure 8. Mean Minimum Sums

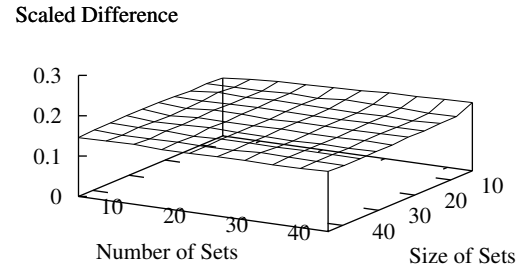

Figure 7. Mean Scaled Differences With Resource Reduction on Each Set

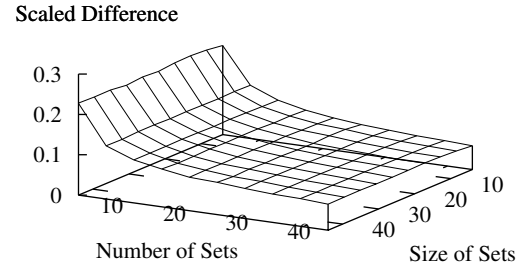

Figure 9. Mean Scaled Differences With Resource Reduction On Number of Sets

The second resource reduction mechanism transforms the outer loop (over `i`) so that it performs only a subset of the iterations, once again using loop perforation so that the loop executes every other iteration. The resulting computation does not consider all of the sets of numbers; it instead considers only every other set. In this case the algorithm may not consider the set with the minimum sum and may compute the index of a set with a larger sum.

Each trial in the simulations fills each of the `m` sets with `n` numbers psuedorandomly selected from the uniform distribution on $[0, 1/\mathtt{n}]$. Note that for any `n`, the maximum possible value for the sum of the numbers in the set is 1; the minimum possible sum is 0. Each simulation runs 10,000 trials for each combination of `m` and `n` such that `m` and `n` are both a multiple of 4 between 4 and 44 (inclusive).

4.2 Absolute Differences

Our first set of simulations applies resource reduction to each set by perforating the inner loop in the model computation. Figure 2 presents, as a function of the number of sets `m` and the size of sets `n` the mean difference (over all trials) between the sum of the set that the resource reduced computation finds and the sum of the set that the original computation finds. The mean differences grow as the size of the sets decreases and (more slowly) as the number of sets increases. The variance of the differences (in Figure 3) exhibits a similar pattern. We attribute this pattern to the fact that as the size of the sets decreases, the difference depends on fewer numbers. The variance in the sum of the numbers in the set is therefore larger, as is the difference between the resource reduced set and the original set. For larger set sizes the mean differences and variances are quite small. This phenomenon reflects that fact that there is a significant correlation between the sum of half of the numbers in the set and the sum of all of the numbers in the set. So while choosing a set based only on a subset of the numbers in the set may cause the algorithm to make a suboptimal choice, the sum of the selected suboptimal set will still be reasonably close to the sum of optimal set.

Our next set of simulations applies resource reduction to the number of sets by perforating the outer loop in the model computation. In this case both the mean differences (in Figure 4) and variances (in Figure 5) both decrease as the number of sets and size of the sets increases. We attribute this behavior to the fact that increasing the size of the set decreases the variance of the sum of the numbers in the set and that increasing the number of sets also decreases the variance in the sums of the sets. So if the resource reduced computation discards the minimum set, increasing the size and/or number of the sets increases the chances that the resource reduced computation will find another set with a minimum close to the minimum found by the original computation. Note that the resource reduced computation for a given number of sets considers the same number of sets as the original computation for half of the number of sets. For larger set sizes and numbers of sets the mean differences are very small.

4.3 Scaled Differences

We next consider not the absolute difference between the minimum values that the resource reduced and original computations find, but the scaled differences — that is, the absolute difference divided by the observed range in the sums of the sets (here the range is the difference between the maximum and minimum sums). Figure 6 presents the mean maximum sum in the collection of sets; Figure 8 presents the mean minimum sum. Note that the two graphs are symmetrical. Note that the distance of the minimum and maximum sums from the expected value of the sum (0.5) increases as the number of sets increases and the size of the sets decreases.

Figure 7 presents the scaled difference for the resource reduced computation on each set (in effect, Figure 2 divided by the difference between Figure 6 and Figure 8). Note that this scaled difference is essentially constant at approximately 15% across the entire range. The scaled differences for the resource reduced computation on the number of sets, on the other hand, decreases as the number of sets increases, with a range between approximately 25% and 7%.

4.4 Application to x264

The model computation provides insight into why x264 tolerates the loop perforation resource reduction mechanism. This mechanism produces a new, more efficient, but less precise metric for evaluating potential matches between reference blocks and the current block to encode. This corresponds to the first resource reduction mechanism in our model computation (which causes the computation to consider only a subset of the numbers in each set). The resulting reduction in the precision of the metric can cause the algorithm to choose a suboptimal reference block for the encoding. But because of redundancy present in the reference blocks (which manifests itself in redundancy in the sums of the numbers used to measure the match at individual points), this suboptimal choice causes only a small reduction in the overall effectiveness of the encoding.

x264 has also been manually coded to work with resource reduction mechanisms that correspond to the second resource reduction mechanism in our model computation (which causes the computation to consider only some of the sets of numbers). Specifically, considering every possible reference block is computationally intractable, so x264 has been manually coded to consider only a subset of the reference blocks. The robustness of the video encoder to considering only a subset of the possible reference blocks corresponds to the robustness of the model computation (see Figures 4 and 9) to considering only some of the sets of numbers.

5. Linked Data Structures

Linked data structures are pervasive in modern software systems. If the nodes in the data structure are allocated dynamically, it is typically impossible, with standard language semantics, to bound the amount of memory the data structure may consume. In extreme cases, the data structure may have memory leaks, typically because it contains references that make nodes reachable even though the data structure will never access the nodes in the future.

5.1 SNMP in Squid

The SNMP module in Squid implements a mapping from names to objects. It stores the mapping in a search tree, with the nodes dynamically allocated as the SNMP module stores new mappings.

It is possible to use cyclic memory allocation to reduce the amount of memory required to store the tree (and eliminate any memory leaks). This allocation strategy sets aside a fixed-size buffer of nodes, then cyclically allocates the data structure nodes out of that buffer [21]. If the data structure requires more nodes than are available in the buffer, this approach will allocate two logically distinct nodes in the same physical node, with the values from the two nodes overwriting each other in the same memory.

In general, the effect of this memory allocation strategy depends on the specific characteristics of the computation. For the SNMP module in Squid, the empirical results show that the computation degrades gracefully. Specifically, forcing the data structure to fit in a fixed-size buffer with node overlaying makes the SNMP module unable to respond successfully to some search queries. But the module continues to execute and successfully respond to a subset of the queries. Moreover, the remainder of the computation remains unaffected and able to continue to provide service to its users.

5.2 The Model Computation

The SNMP search tree is a linked, null-terminated, acyclic data structure. We use a simple example of such a data structure, a singly-linked list, to explore the potential consequences of subset memory allocation strategies that reallocate live memory. The linked list implements a set of values, with the allocation taking place when the list inserts a new value into the set. We consider two implementations of this operation: one that inserts the value at the front of the list and one that inserts the value at the end of the list. The usage scenario has the client repeatedly inserting values into the set, interspersing the insertions with lookups.

```
typedef struct node {
  int value;
  struct node *next;
};
struct node *first, *newNode();
void prepend(int v) {
  struct node *n = newNode();
  n->value = v; n->next = first;
  first = n;
}
void append(int v) {
  struct node *n = newNode();
  if (first == NULL) first = n;
  else {
    l = first;
    while (l->next != NULL) l = l->next;
    l->next = n;
  }
  n->value = v; n->next = NULL;
}
int lookup(int v) {
  struct node *l = first;
  while (l != NULL)
    if (l->value == v) return 1;
  return 0;
}
```

We consider various reduced resource allocation mechanisms, all of which allocate new list nodes out of a fixed buffer of nodes. We note that all of the mechanisms we consider may cause `newNode` to return a node that is already in the list. Because a `prepend` operation applied to a node already in the list creates a cyclic list and causes subsequent `lookup`s of values not already in the list to infinite loop, our further analysis only considers `append` operations.

5.2.1 Least Recently Allocated Allocation

The first allocation mechanism is cyclic allocation, which returns the least recently allocated node in the buffer in response to each node allocation request (in other words, the allocator always returns the least recently allocated node). When presented with a sequence of `append` operations, this mechanism builds up a standard list. When the allocation mechanism wraps around, it returns the first element in the list and the assignment `n->next=NULL` truncates the list. The result is a list containing only the most recently inserted element. Successive `append` operations build up a list containing the most recently inserted elements. Over time, given an unbounded number of insertions, the list contains, on average, the most recently inserted $b/2$ values (where b is the number of nodes in the buffer).

5.2.2 Most Recently Allocated Allocation

The next allocation mechanism always returns the most recently allocated node (i.e., the last node in the buffer) once all of the nodes in the buffer have been allocated once. With a sequence of `append` operations, the effect is simply to replace the last value in the list (the one most recently inserted) with the new value. After b insertions, the list contains the

first $b-1$ inserted nodes and the last inserted node (where b is the number of nodes in the buffer).

5.2.3 Random Allocation

The next allocation mechanism returns a random node in the buffer. Over time, given an unbounded number of insertions, the list contains, on average, approximately $0.125b$ nodes (where b is the number of nodes in the buffer). We obtained this number through a simulation which repeatedly inserts nodes into the list, psuedorandomly selecting one of the nodes in the buffer as the node to insert, then measuring the length of the list after each insertion to compute the mean length of the list during the simulation (which is approximately $0.125b$). Note the relatively low utilization of the available nodes in the buffer — ideally, the list would contain all b of the nodes in the buffer. The reason for the low utilization is that when the allocation algorithm selects a node that is already in the list, the insertion truncates the list at that node. So as the list grows in length, insertions that shorten the list become increasingly likely.

5.2.4 Discussion

Data structure consistency is a critical acceptability property [10, 26]. Reduced resource allocation strategies and null-terminated linked data structures work best with algorithms that append null-terminated nodes to the leaves of the data structure — other strategies may cause cycles that leave the data structure inconsistent. The linked list data structure in our model computation is one example of a data structure that appends at the leaves (in this case, simply at the end of the list). More complex examples include binary search trees and separately-chained hash tables that store all elements that hash to the same location in the table together in a linked list. Like linked lists, these data structures can be used to implement a variety of abstractions such as sets and maps.

We note that for linked lists and data structures (like separately-chained hash tables) that incorporate linked lists, inserting only at the leaves can complicate strategies that attempt to reduce lookup times by keeping the lists sorted — standard insertion algorithms for sorted lists involve insertions in the middle of the list.

5.3 Usage Scenarios

To date, the most compelling usage scenarios for reduced resource allocation involve the elimination of memory leaks, which can threaten the survival of the entire system [21]. The primary drawback of the resource reduction mechanism is the inadvertent elimination of elements that would, with standard allocation mechanisms, remain present in the data structure. We next outline several scenarios in which the benefits of reduced resource allocation can outweigh the drawbacks.

One scenario occurs when the application does not need the data structure to necessarily contain all of the inserted elements — for example, the application may use the data structure to cache previously fetched elements [21]. In this case reduced resource allocation may degrade the effectiveness of the cache, but will leave the application capable of delivering its full functionality to its users. Another scenario occurs when the computation can tolerate the loss of elements that typically occurs with reduced resource allocation. The use of reduced resource allocation in the SNMP module, for example, degrades the delivered functionality of the SNMP module (the SNMP module does not correctly process some queries), but leaves the remaining Squid functionality unimpaired. We see this general pattern, applying reduced resource allocation to eliminate a problem within one module that would otherwise threaten the survival of the entire system, as a particularly compelling usage scenario.

5.4 Reduced-Resource Aware Implementations

We have so far assumed that the developer of the data structure manipulation algorithms is oblivious to the potential application of resource reduction mechanisms and that the implementation is therefore coded only to consider cases that arise with standard semantics. But it is of course possible to develop algorithms that are purposefully designed to work with resource reduction mechanisms. Potential implementation strategies could include:

- **Relaxed Invariants:** Implementing the data structure manipulation algorithms to work with data structures that have a relaxed set of invariants. For example, a data structure implementation coded to operate in the presence of cycles would enable the use of resource reduction algorithms that insert items at internal points in lists and trees (as opposed to at the leaves).
- **Data Structure Repair:** Instead of coding every data structure manipulation operation to work with relaxed data structure invariants, it is possible to instead simply use data structure repair [8–10] to restore the desired invariants at appropriate points (typically the beginning or end of data structure operations). This approach would make it possible for the vast majority of data structure operations to be coded to operate on data structures that satisfy all of the stated data structure consistency properties.
- **Explicit Removal:** Instead of simply reusing allocated data structure nodes (which introduces the possibility of conceptually dangling references to and/or from the previous location of the node in the data structure), the data structure implementation could be coded to explicitly remove all reused nodes from their previous location before inserting them into the new location. This approach would preserve all of the standard invariants and eliminate the inadvertent deletion of nodes originally reachable via the reallocated node when it was in its original position. This strategy would also maximize the achiev-

able utilization of the available space — for most linked data structures at capacity, this strategy would leave all of the nodes linked into the data structure and accessible via standard data structure operations. Specifically for the linked list in our model computation, the list would (after the insertion of at least b values) always contain the maximum b values.

5.5 Fixed-Size Tables

In this section we have focused on linked data structures, which are often deployed because they enable applications to adjust their memory usage to the characteristics of different inputs as different inputs elicit different memory usage behavior from the application (the application typically uses less memory to process small inputs and more memory to process large inputs). But it is also possible to apply resource reduction mechanisms to fixed-sized tables. These data structures are typically implemented as a single block of memory allocated at the start of the computation. They are designed to store at most a fixed number of items determined by the size of the memory block. If the computation attempts to insert an item into a table that is already full, it typically halts with an error indicating that a larger-sized table is required to process the current input. One example of such a data structure is an open-addressed hash table (which stores all of the values or key/value pairs directly in the hash table buckets, using probing techniques to resolve collisions).

One resource reduction mechanism overwrites old values with new values or discards new values when it is impossible or inconvenient to store them both. For example, one strategy would simply overwrite the least recently used or inserted table entry with the new entry when the client attempts to insert a new entry into a table that is already full. In some circumstances it may be convenient to apply such mechanisms even when the table is not full. For example, if two values or key/value pairs hash to the same bucket, the implementation may simply discard one of the values or key/value pairs rather than apply techniques such as linear probing, quadratic probing, double hashing, or converting the bucket in question into a separately-chained bucket. The potential advantages of avoiding these more complex techniques include simplicity, convenience, efficiency, and even correctness (a simpler implementation provides the developer with fewer opportunities to introduce errors). For example, an out of bounds addressing error introduced during an attempt to implement a more sophisticated response to events such as hash table collisions can (depending on how the application uses the table) easily have more serious consequences than overwriting existing table entries or discarding new table entries.

Note that such mechanisms eliminate the requirement that the fixed-size table be able to store all of the items that client attempts to insert into the table. They may therefore make it possible to make the table smaller because it does not need to be sized for the largest anticipated usage scenario — it can instead be sized for more common-case scenarios while providing successful execution with reduced functionality for less common scenarios that exceed the table size. We note that current approaches often involve the use of fixed-size tables that (if they abort when the client attempts to take them beyond their capacity) threaten the survival of the entire computation. Resource reduction mechanisms can therefore enhance the robustness and resilience of the system by enabling the table to execute successfully through otherwise fatal events such as attempting to insert a new item into a table that is already full.

6. Redundancy and Correlation

All of our resource reduction mechanisms exploit redundancy in the values that they manipulate. One way to see this redundancy is to examine correlations between the values. Resource reduction works well with the mean pattern because (for most probability distributions) the mean of a subset of numbers is highly correlated with the mean of the set of numbers itself. Similarly, the sum of a subset of numbers is correlated (after extraplotation) with the sum of the set. And, the minimum of a subset of numbers is correlated with the minimum of the set. The correlations in all of these examples are relatively easy to see because all of the examples perform an aggregation or reduction operation. The redundancy that appears in the original set of values is the root cause of the correlation between the results of computations on subsets of the original set of values.

Of course, there are other forms of redundancy and correlation that make computations amenable to resource reduction. Consider, for example, standard iterative computations (such as differential equation solvers), that converge to an acceptably accurate answer. As the computation converges, successive iterations compute highly correlated values. This correlation is the reason that the standard resource reduction mechanism for converging iterative computations (terminating the iteration before it converges) works well in practice.

Internet search engines also compute correlated values — the results for a given query are highly correlated, which enables resource reduction mechanisms (for example, searching only a subset of the documents [6]) that cause the search engine to return only a subset of the results that the original computation would have returned. This example generalizes to virtually all information retrieval computations.

As another example, the sets of moves that automated game-playing algorithms consider are often of roughly equivalent quality – in other words, the effects that the different moves have on the outcome of the game are highly correlated. This correlation enables the successful application of resource reduction mechanisms that force the algorithm to execute with reduced memory by allocating conceptually distinct data in the same memory. While the reduced resource version does not generate the same moves as the original version, it exhibits roughly equivalent competence [21].

Finally, the values that applications communicate over a network are often correlated. One simple way to exploit this fact is to use short-circuit communication — instead of performing a standard request/response interaction, memoize the results of previous interactions and return the response whose request most closely matches the current request. It is also possible to similarly memoize direct communications that do not occur as part of a request/response interaction [26].

7. Finding New Patterns

We anticipate that, as the field progresses, researchers and practitioners will find other computational patterns that work well with reduced resource computing mechanisms. We propose several strategies for finding such patterns:

- **Empirical Testing:** We obtained our current set of patterns by applying resource reduction mechanisms broadly across selected benchmark applications, empirically finding computations that responded well to these mechanisms, then analyzing the computations to understand the reasons why they responded so well. We anticipate that applying existing and new resource reduction mechanisms across a broader set of benchmark applications would reveal additional patterns that work well with such mechanisms.
- **Analytical Analysis:** Another approach reverses the process — it starts with simple computations (such as the mean, sum, and minimum computations) that compute correlated values when applied to correlated data sets such as subsets of values drawn from the same set or probability distribution. As the results in this paper show, such computations often respond well to resource reduction mechanisms. The next step would be to find instances of such computations in existing applications, then evaluate (either empirically or analytically) the suitability of resource reduction as applied to these computations in the context of the application.
- **Correlated Values:** To date, every successful application of reduced resource computing mechanisms has exploited a form of redundancy that shows up as correlations in the values that various computations compute. Looking for correlations between computed values may therefore be a productive way to find other patterns that interact well with resource reduction. This could be done either empirically, by examining computed values to look for correlations, or analytically.
- **Reduction Operations:** Three of our four patterns apply reduction operations (sum, mean, min) that reduce a set of values to a single value. We anticipate that other reduction operations would be good candidates for the application of resource reduction mechanisms.

We also anticipate that, as the benefits of resource reduced computing become more widely known, developers will begin to consciously engineer systems to use computation patterns that interact well with reduced resource computing mechanisms. Over time, systems will become increasingly engineered to work productively with these mechanisms and other modern program transformations that may change the semantics of the program in return for benefits such as increased robustness, resilience, security, memory efficiency, and performance [4, 19, 21, 22, 26, 27, 32, 33].

8. Criticality Testing

In our experience, it is possible to apply resource reduction successfully to only a subset of the subcomputations in a given computation. Indeed, the inappropriate application of resource reduction can easily cause applications to crash, take longer to execute, or produce wildly inaccurate results [16, 20, 23]. Examples of unsuitable computations that we have encountered in practice include tasks or loops that allocate subsequently accessed objects, efficient prefilters that eliminate objects unsuitable for subsequent more involved processing (applying resource reduction to the prefilter can decrease the overall performance of the application), and tasks or loops that produce the output of the application or drive the processing of input units (for example, applying resource reduction to the loop that iterates over the video frames in X264 unacceptably causes X264 to drop frames).

We therefore advocate the use of *criticality testing*, which automatically applies resource reduction across the application, using a test suite of representative inputs and an appropriate accuracy metric to automatically find subcomputations that work well with resource reduction [16, 20, 23]. One can view the resource reduction mechanisms as automatically generating a search space of applications surrounding the original application, with the accuracy metric enabling the search algorithm to explore the space to find variants with more desirable performance/accuracy characteristics for the current usage context than the original application.

An alternative approach requires the developer to provide multiple implementations of different subcomputations, potentially with different performance and accuracy characteristics [1, 2]. A search algorithm can then explore various implementation combinations to find desirable points in the performance/accuracy trade-off space. A claimed advantage is that it is less likely to deliver applications that can produce unanticipated results. A disadvantage is the developer effort required to find appropriate subcomputations and develop alternate versions of these subcomputations.

We anticipate that, in practice, developers may often encounter some difficulty in finding and developing alternate versions of subcomputations with good accuracy/performance characteristics in large applications, especially if the op-

timization effort is separate from the initial development effort and the developer starts out unfamiliar with the application. In typical scenarios we believe the most productive approach will involve the automatic application of resource reduction mechanisms in one (or more) of three ways:

- **Profiling:** Automatically applying resource reduction to subcomputations, then observing the resulting accuracy and performance effects, to help developers identify suitable subcomputations (subcomputations with desirable accuracy and performance characteristics after resource reduction) that they then manually optimize [20].
- **Developer Approval:** Instead of manually optimizing the identified suitable subcomputations, the developer can simply examine the resource reduced subcomputations and approve or reject the resource reduced versions for production use. This scenario preserves the developer involvement perceived to promote the safe use of resource reduction, but eliminates the need for the developer to modify the application.
- **Automatic Application:** The direct application of resource reduction without any developer involvement with the source code of the application. In these scenarios performance and accuracy results from running the application on representative inputs provide the primary mechanism for evaluating the acceptability of the resource reduction. This approach may be particularly appropriate when developers that can work with the application are unavailable, when developers are a scarce resource that can be more productively applied to other activities, or when the source code of the application is not available. We note that it is relatively straightforward to apply the specific resource reduction mechanisms presented in this paper directly to compiled binaries, eliminating the need for source code access.
- **Static Analyses:** We anticipate the development of static program analyses that will automatically recognize patterns (such as those identified in this paper), or, more generally, computations, that interact well with resource reduction mechanisms. In combination with either simulations (such as those presented in this paper) or the development of symbolic analytical models (or a combination of the two), these static analyses would make it possible to precisely characterize the impact on the accuracy (and potentially the performance) of applying different resource reduction mechanisms to the specific analyzed application at hand. The simulations and models could be tailored to the recognized pattern and its usage context within the application. It would also be possible to draw on an existing set of simulation results and/or analytical models. And of course combinations of these two approaches are also possible.

9. Related Work

The resource reduction mechanisms discussed in this paper were all developed in previous research [16, 20, 21, 23, 24]. This paper identifies general computational patterns that interact well with these mechanisms, presents concrete manifestation of these patterns in the model computations, and uses simulation to quantitatively explore the impact of resource reduction.

There is a long history of developing application-specific algorithms that can execute at a variety of points in the performance versus accuracy trade-off space. This paper, in contrast, focuses on general, broadly applicable resource reduction mechanisms that can automatically enhance applications to productively trade off accuracy in return for performance, even though the applications may not have been originally designed to support these kinds of trade offs.

9.1 Memory Reduction Mechanisms

In this paper we have considered two kinds of resource reduction mechanisms: mechanisms that reduce the amount of memory and mechanisms that reduce the amount of computation. To the best of our knowledge, the first (and, to date, only) proposal for a general memory reduction mechanism was cyclic memory allocation [21]. It is also possible to use out of bounds access redirection (cyclically or otherwise redirecting out of bounds accesses back within the accessed memory block) [27, 29, 30] to reduce the size of tables or arrays (or, for that matter, any data structure allocated in a contiguous block of memory). The out of bounds redirection eliminates any cross-block memory corruption or memory error exceptions that might otherwise threaten the survival of the application after reducing the size of the memory block holding the table or array.

9.2 Computation Reduction Mechanisms

To the best of our knowledge, the first proposal for a general computation reduction mechanism was discarding tasks [23, 24]. The proposed uses included surviving errors in tasks, eliminating barrier idling, and using the automatically derived timing and distortion models to navigate the induced performance versus accuracy space. The models make it possible to either maximize accuracy subject to specified performance constraints, to maximize performance subject to specified accuracy constraints, or to satisfy more complex combinations of performance and accuracy optimization criteria.

Many of the discarded tasks in the benchmark applications simply execute blocks of loop iterations. This fact makes it obvious that discarding iterations of the corresponding loops (i.e., loop perforation) would have the same effect on the performance and accuracy as discarding the corresponding tasks. One advantage of discarding loop iterations instead of tasks is that it is possible to apply the corresponding loop perforation transformations directly to a

broad range of programs written in standard programming languages [16, 20] (as opposed to programs written in languages with a task construct).

9.3 Manual Versus Automatic Techniques

The resource reduction mechanisms in this paper were all initially developed for largely automatic application, with criticality testing typically used to find subcomputations that can profitably tolerate resource reduction. The PetaBricks and Green systems, in contrast, require the developer to create the trade-off space by designing and implementing multiple implementations of the same subcomputation [1, 2]. As described in Section 8, the automatic application of resource reduction can significantly reduce the developer effort required to generate and find desirable points within this trade-off space. Moreover, automatic mechanisms such as loop perforation and discarding tasks can discover latent desirable trade-offs that are simply unavailable to human developers:

- the developer may not suspect that a profitable trade-off exists in a given subcomputation,
- the developer may not understand the subcomputation well enough to realize that a profitable trade-off may be available,
- the developer may not have the time, expertise, or knowledge required to provide another implementation of the subcomputation, or
- the developer may simply be unaware that the subcomputation exists at all.

Finally, automatic techniques can even enable users with no software development ability whatsoever to obtain new versions of their applications that can execute at a variety of different points within the underlying performance versus accuracy trade-off space that resource reduction mechanisms automatically induce.

9.4 Dynamic Control

It is possible to apply resource reduction mechanisms such as loop perforation [16] and discarding tasks [23] dynamically to move a running application to different points in its underlying performance versus accuracy trade-off space. It is also possible to convert static configuration parameters into *dynamic knobs* — dynamic control variables stored in the address space of a running application [17]. Like resource reduction mechanisms, dynamic knobs make it possible to move a running application to different points in the induced performance versus accuracy trade-off space without otherwise perturbing the execution.

These mechanisms make it possible for an appropriate control system to monitor and dynamically adapt the execution of the application to ensure that the application meets performance or accuracy targets in the face of fluctuations in the amount of resources that the underlying computational platform delivers to the application [16, 17]. Such control systems typically work with a model that characterizes how the application responds to different resource reduction mechanisms or dynamic knob settings. Ideally, the control system would guarantee good convergence and predictability properties such as stability, lack of oscillation, bounded settling time, and known overshoot [17]. We have developed controllers with some or all of these properties [16, 17] — in particular the PowerDial control system [17] has all of these properties. These controllers are designed for applications (such as video encoders) that are intended to produce a sequence of outputs at a regular rate with a target time between outputs. These controllers use the Application Heartbeats framework [15] to monitor the time between outputs and use either loop perforation [16] or dynamic knobs [17] to control the application to maximize accuracy while ensuring that they produce outputs at the target rate.

It would also be possible to combine manual techniques (such as those supported by PetaBricks or Green) with dynamic controllers. PetaBricks does not have a dynamic control component [1]. Green uses heuristic control to manage accuracy but does not control or even monitor performance [2]. Green's control system is also completely heuristic, with no guaranteed convergence or predictability properties whatsoever.

9.5 Unsound Program Transformations

Reduced resource computing mechanisms are yet another example of unsound program transformations. In contrast to traditional sound transformations (which operate under the restrictive constraint of preserving the semantics of the original program), unsound transformations have the freedom to change the behavior of the program in principled ways. Unsound transformations have been shown to enable applications to productively survive memory errors [4, 27, 32], code injection attacks [22, 27, 32, 33], data structure corruption errors [8–10], memory leaks [21], and infinite loops [21]. Like some of the reduced resource computing mechanisms identified in this paper, unsound loop parallelization strategies have been shown to increase performance at the cost of (in some cases) potentially decreasing the accuracy of the computation [19]. Finally, reduced resource computing mechanisms can help applications satisfy acceptability properties involving the amount of resources required to execute the computation [26].

10. Conclusion

Dealing with limited resources is a critical issue in virtually every aspect of life, including computing. The standard approach to dealing with this fact is to rely on developers to produce algorithms that have been tailored by hand for the specific situation at hand.

Over the last several years researchers have developed a collection of simple, broadly applicable mechanisms that en-

able computations to execute successfully in the presence of reduced resources. This development promises to dramatically simplify the process of obtaining computations that can operate successfully over the wide range of operating conditions that will characterize future computing environments.

10.1 Value Systems

All of these resource reduction mechanisms go against the basic value system of the field in that they may change the result that the computation produces — in the traditional (and, in our view, outmoded) terminology, they are unsound. Through its identification of general patterns and model computations that interact well with these mechanisms and the statistical properties of these mechanisms, this paper attempts to enhance our understanding of the underlying computational phenomena behind the observed empirical success of these mechanisms. In this way, this paper transcends the traditional use of discrete formal logic to characterize application behavior in terms of rigid, fixed input/output relationships. It instead promotes a new, more flexible model that incorporates continuous statistical relationships between not just the input and output, but also the amount of resources that the computation consumes to produce the output.

10.2 Benefits

We anticipate several benefits that can stem from this enhanced perspective. The first is a greater comfort with and acceptance of resource reduction mechanisms, which, in turn, can make the benefits they provide more broadly available. The second is that it may enable developers to more easily recognize and use computational patterns that interact well with these mechansims. The third is that it may inspire the development of additional resource reduction mechanisms and the identification of both existing and new computational patterns that work well with these mechanisms.

A final benefit is that it can broaden the scope of the field itself and promote its movement away from an outmoded, regressive value system that is increasingly hindering its development. Specifically, this research promotes the development and evolution of software systems via the automated exploration of automatically generated search spaces rather than full manual development. The evaluation mechanism is empirical, based on observing the behavior of the system as it executes on representative inputs, rather than theoretical and based on an analysis of the text of the program before it runs. The goal is to deliver flexible programs that usually produce acceptably accurate output across a broad range of operating conditions rather than attempting to develop rigid, inflexible programs that satisfy hard logical correctness conditions when run on perfect execution platforms. In this way the research points the direction to software that can operate successfully in the increasingly dynamic and unpredictable computing environments of the future.

Acknowledgements

We would like to thank the anonymous reviewers for their useful criticism, comments, and suggestions. This research was supported by DARPA Cooperative Agreement FA8750-06-2-0189, the Samsung Corporation, and NSF Awards 0811397, 0835652, and 0905244.

References

[1] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *PLDI '09*.

[2] W. Baek and T. Chilimbi. Green: A system for supporting energy-conscious programming using principled approximation. Technical Report TR-2009-089, Microsoft Research, Aug. 2009.

[3] J. Berg, J. Tymoczko, and L. Stryer. *Biochemistry*. W.H. Freeman, 2006.

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

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*.

[6] E. Brewer. Towards robust distributed systems. Keynote, ACM Symposium on Principles of Distributed Systems (PODC), 2000.

[7] 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.

[8] B. Demsky, M. Ernst, P. Guo, S. McCamant, J. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA '06*.

[9] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA '03*, 2003.

[10] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05*, 2005.

[11] R. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium in Applied Mathematics*, number 19, 1967.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, 1990.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969.

[15] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *ICAC'10: 7th International Conference on Autonomic Computing*, 2010.

[16] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using Code Perforation to Improve Performance,

Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT, Sept. 2009.

[17] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Power-Award Computing with Dynamic Knobs. Technical Report TR-2010-027, Computer Science and Artificial Intelligence Laboratory, MIT, Aug. 2010.

[18] D. Le Gall. MPEG: A video compression standard for multimedia applications. *CACM*, Apr. 1991.

[19] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report TR-2010-038, Computer Science and Artificial Intelligence Laboratory, MIT, Aug. 2010.

[20] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard. Quality of service profiling. In *ISCE '10*.

[21] H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM '07*.

[22] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP '09*.

[23] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06*.

[24] M. Rinard. Using early phase termination to eliminate load imbalancess at barrier synchronization points. In *OOPSLA '07*.

[25] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford University, Department of Computer Science, 1994.

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

[27] 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 '04*.

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

[29] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC 04*.

[30] M. C. Rinard, C. Cadar, and H. H. Nguyen. Exploring the acceptability envelope. In *OOPSLA '05 Companion*.

[31] K. Schmidt-Nielsen, R. Schroter, and A. Shkolnik. Desaturation of exhaled air in camels. *Proceedings of the Royal Society of London, Series B, Biological Sciences*, 211(1184), 1981.

[32] 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*.

[33] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX '05*.