Increasing Confidence in Proper Execution through Invariant Checking

Miryung Kim and Andrew Petersen
{miryung, petersen}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington

Abstract

With the size of processors and the distance between components decreasing rapidly, the probability of transient errors occurring during execution is increasing. While these errors can cause any program to fail or behave erratically, such behavior is not acceptable in many domains, including safety-critical and high-reliability systems. In simulations that take days or weeks to complete or control systems that should not fail, for example, the ability to prevent a serious system failure can be very beneficial. One possible step towards ensuring correct processor execution involves self-verification of the execution of groups of instructions using an invariant that should hold before and after a block is executed.

Invariants, provided by the programmer or by an automatic invariant detector, can be used to increase confidence in the correctness of the primary computations and will form the basis for this self-verifying architecture. If the processor's output does not preserve the invariant, the processor will stop execution and can be "rolled back" to a previously verified state to re-execute the program from that point. While not guaranteeing absolute correctness of execution, the additional tests catch many of the processor's errors and can reduce the chance of a serious fault, with the effectiveness and performance impact of the method depending on the strength (and exhaustiveness) of the invariants used.

1 Introduction

Fault tolerant architectures have become more interesting in recent years, as it has become increasingly evident that the small, high speed processors of the future will be extremely prone to transient errors caused by electronic noise and random natural interference. This problem is compounded by the huge costs that will have to be borne to fully verify the correctness of increasingly complex and hard to design systems. Since not developing (or not using) the advanced systems of the future is not an option, a solution will have to be found to make systems more tolerant (or at least cogent) of faults caused either by a fault in design or by a random event.

To that end, we propose a self-verifying system where the compilers insert checks to verify that a user-provided or automatically generated invariant holds at that point in the program's execution. In general, these invariants will not verify specific, individual statements. Instead, they will correspond to groups of statements, so performance should not be greatly affected. Invariant checking can be performed more quickly than we can re-compute all of the results, but it does incur some performance cost above not checking the results at all. Thus, while this method does not guarantee the absolute correctness of the program or the processor's results, it does greatly enhance our confidence in the result's validity by guaranteeing correctness with respect to the invariants, and it does so without very little impact to performance. Our work is based on an idea by Jeong and Jamison [6] that checks invariants with one processor of a dual processor machine. Our design does not incur the extra communication and control overhead that an extra processor brings, but it does sacrifice the ability to catch faults in the processor's design, since a secondary processor would not necessarily include the same design flaws as the primary processor.

In this paper, we concentrate on how to implement a compiler that supports a self-verifying architecture. We hope that by gathering data on the relative performance penalties of various methods for inserting invariants, we will gain some insight into what technology will be needed to build a compiler for a self-verifying architecture and how best to implement it.

In Section 2, we provide a brief overview of invariants, their traditional use, and current invariant verification technology. Next, in Section 3, we examine the overall structure of a self-verifying architecture, and then in Section 4, discuss the metrics we use to measure the effectiveness of our technique. This leads us to introduce the process we used for adding invar-

iants to programs, describe our test programs, discuss the reasoning behind our choices, and examine results obtained in a simulation using SimpleScalar v3.0 [3] in Sections 5 and 6. Finally, Section 7 introduces related work, and we close with a summary and possible future work in Section 8.

2 Invariants

Invariants, or relationships between data members that hold for all possible values of the members involved, are commonly used to enforce program safety (especially type safety). Programmers are often encouraged to annotate their programs with invariants, with the aim of improving maintainability and ease of verification. While not used universally, invariants are commonly employed, and when explicitly stated, are thought to make it easier to design and implement programs.

Therefore, the software engineering community has exerted a great deal of effort towards building sound automatic invariant generators and checkers, to make the creation of invariants less manpower intensive and the types of invariants included in programs more uniform. However, at the moment, most invariant generation is done manually. If their annotation is already performed during development, this processor verification technique should not require additional work on the part of the programmer. However, if it is not already part of the development process, we hope to encourage its use during the design and testing process without adding to the developer's workload. With the facilitation of the these goals in mind, we tried to select an automatic invariant detector that will use any provided annotations while requiring a minimum of user input and that could be used in an automated process for inserting invariant checks into code.

The available tools can be roughly divided into two basic groups: static checkers and dynamic detectors. Static invariant checkers need not run the program but must be guided by human input and as such, often require annotations or specifications created by the program designer or implementer [5]. Dynamic checkers, in contrast, usually do not require additional input from a human (although they will accept and use annotations if available) but must execute the program being analyzed with a large test suite to infer possible invariants. Unfortunately, while the two groups differ in the amount of user input required, all current invariant detectors have several shortcomings due to the intrinsic difficulty of invariant generation.

First, invariant detectors cannot discover a complete set of invariants, because the problem of determining all invariants is undecidable. Second, invariant generation tools operate at the level of functional granularity, partly because the idea for invariants developed from Hoare triples and loop invariants, which are derived from methods that consider basic blocks. Third, invariant detectors only suggest relationships between the parameters passed to the function and the return value, because they generally begin their search by looking at those values and examine transformations performed on them. Hence, invariant detectors may fail to suggest an interesting invariant that involves local data allocated in the callee's stack. Finally, it is almost always necessary to have substantial amounts of input from the programmer to direct the checker towards useful invariants.

Despite these shortcomings, we chose Daikon [5] as our invariant detector. To direct it toward interesting invariants, we provided each program with a test suite, since Daikon is a dynamic invariant checker. In addition, when Daikon fails to generate invariants of interest, we enhanced the its output by manually adding invariants.

3 A Self-Verifying Architecture

Since we perform invariant checks on the same processor that performs the main computations, our model for a self-verifying architecture does not need an additional computation unit. Therefore, the compiler technology we espouse should be able to run on any system without additional cost. Thus, detecting errors and terminating execution if an error is found is cheap, in terms of additional design and hardware costs. (We will investigate performance penalties in the next three sections.) However, the ability to roll back the processor to an earlier state if an invariant is violated will require additional hardware to store a state and commit the changes once an invariant check is passed. In addition, some control is needed to avoid an infinite loop that might occur if the program (or invariant) is flawed, since our method cannot determine whether an error is caused by a processor fault or a fault in the program itself. We do not investigate the costs involved with designing or integrating special hardware or in

using exceptions that will return the processor to the last successfully checked state, but these ideas are definitely targets for future work.

For the purposes of this project, if an error is found, the value of a static, global variable is changed to signal that an error has been detected and execution continues. This avoids the long jump necessary to throw an exception. To verify that invariants hold, checks are inserted as simple if-statements. If any data (or state) needs to be stored to perform a check later in the program, a new variable is created, and the required data is stored in it as soon as possible after an invariant check, to increase the chance that the data is correct. Often, this means that a series of new variable declarations and initializations occur at the beginning of functions.

4 Metrics

Our goal while building this self-verifying technique is to determine how much confidence is gained in our belief that the program executed correctly. However, this gain must be mitigated by the amount of performance was sacrificed. In Section 4.1, we discuss the idea of "confidence gain" by describing how processor reliability relates to instruction executions. In Section 4.2, we discuss what we mean by "performance sacrifice" and introduce a metric that measures it. Finally, in Section 4.3, we explain how we use the metrics from the two previous sections to interpret our experimental results.

4.1 Confidence Gain

Unfortunately, we cannot guarantee that a program has executed correctly, since the problem of absolute program correctness is undecidable. (To do so would mean we have solved the halting problem!) However, we can increase the user's confidence in the correctness of any results that are obtained from the program.

While considering how to measure how effective inserting invariant checks were, we thought about how much confidence is gained when a single computation is repeated. Assume that the ratio of correct to total computations for a processor is X. Then, our initial confidence in the computation is X. If we repeat this computation once more as a check, there are four possibilities: the processor computes both instructions correctly, incorrectly computes the check, incorrectly computes the first instruction, or incorrectly computes both instructions. Thus, the chance that we will detect some error is at least $1 - X^2$, since any of the first three cases will always lead to the detection of the error. (It is possible that the last case will also reveal an error, but we will assume that this is not so.) Hence, for an arbitrarily high confidence, we need only repeat the computation some limited number of times. Therefore, any confidence gain that we obtain must be at less cost than re-executing the program some number of times.

The best way to measure confidence gain is to prove that invariant checking guarantees the program is correct. Since we are unable to generate a complete set of invariants and it is difficult to check by how much confidence increases when invariant checks are applied, we approximate confidence gain with the ratio of the number of instructions added to the program to the number of instructions in the original program. This is an extremely conservative estimate, since it assumes that each instruction that is added checks only one instruction from the original program and the idea behind invariants is to check the "intent" of a block of code which may contains a large number of instructions. However, it gives us a basis from which to begin.

$$Instruction\_increase\_ratio = \frac{\#instructions\_after\_invariants - \#instructions\_before}{\#instruction\_before}$$

Figure 1. Confidence Gain Heuristic

In Figure 1, we introduce our heuristic for estimating confidence gain, the *Instruction Increase Ratio*: This metric measures the increase in the number of instructions after adding invariant checks. Because it only counts instructions added for invariant checking purposes (which are used only to verify the processor), we can use this metric to approximate the confidence gained by invariant checking.

The second metric we introduce is the *Computation instruction increase ratio*, which is revealed in Figure 2. It compares the number of non-memory instructions added by invariant checks to the total number of non-memory instructions in the original program. Because reference instructions do not contribute to our verification computations, it may be more accurate to remove memory-referencing instructions from our confidence gain metric. Thus, we use *Computation instruction increase*

ratio as our measure of net confidence gain when using invariant checks.

$$computatio\ n\_instructio\ n\_increase\_ratio = \frac{\#Non\_memory\_reference\_instructio\ ns\_after\_in\ var\ iants - \#Non\_memory\_reference\_instructio\ ns\_before}{\#Non\_memory\_reference\_instructio\ n\_before}$$

Figure 2. Net Confidence Gain Heuristic

4.2 Performance Sacrifice

The term "performance sacrifice" refers to the cost paid to check the invariants added to the program. The *Cycle time increase ratio* stands for the cycle time increase caused by the addition of invariant checks. (Refer to Figure 3.) Because it uses the difference in cycle time between the instrumented and original programs, it can be used as a metric for the performance sacrifice required to perform invariant checks.

$$Cycle\_time\_increase\_ratio = \frac{Cycle\_time\_after\_in\ var iants - Cycle\_time\_before}{Cycle\_time\_before}$$

Figure 3. Performance Sacrifice Metric

4.3 Interpretation of Metrics

Given that we have an original program P and several version of the same program P(i) that have differing numbers of invariant checks, we can use the metrics from above to measure the effectiveness and efficiency of the checks we have added. Note that later in the paper, we will use increasing values of i to denote increasing numbers of invariant checks added.

As above, the performance sacrifice is represented using the *Cycle time increase ratio* and confidence gain is represented using the *Instruction increase ratio*. By plotting the *Cycle time increase ratio* for different levels of invariants on the same program, we can also see how quickly the performance sacrifice changes as invariants of differing complexities are inserted. In the same way, we can measure how quickly confidence gain changes as the number and degree of invariants added differ. These two rates can be used to see whether monitoring the type and number of invariants added to a program will be important for controlling the amount of performance sacrificed.

For example, if we find that for a certain type of application, the *cycle time increase ratio* increases dramatically faster than the *instruction increase ratio*, then we will know that controlling invariant additions is critical for maintaining a bearable performance. However, if the reverse holds, then we will know that a larger number of invariants can be added without adversely affecting performance to any great extent.

5 Measurements

Before we investigate the performance overhead involved when inserting and checking invariants, we focused on finding an optimal granularity for inserted invariants (Section 4.1), discovering the best location for inserting invariants (Section 4.2), and utilizing the most optimal spatial and temporal complexity for checking invariants (Section 4.3). These work was needed because we had to find the appropriate way of inserting invariants because adding invariants in different style could make big difference in performance gain.

5.1 Invariant Granularity

As we mentioned in Section 2, most invariant generators suggest invariants at the functional level of granularity. Despite the state of the current invariant detection technology, we wanted to contrast the performance differences between invariants derived at the functional level and invariants obtained at the statement level.

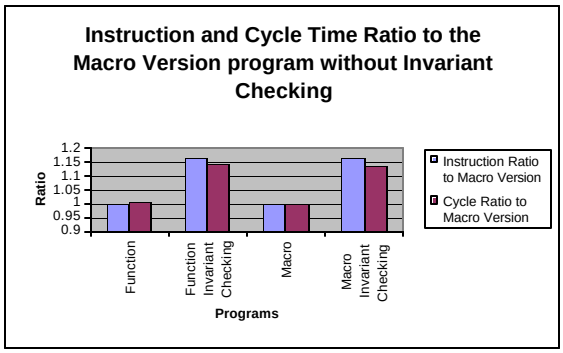

Figure 4. Instruction and Cycle Time Ratios: the same invariants were inserted in both cases to make the comparison fair.

Figure 4 shows that if we use in-line invariants with in-line functions, the least amount of performance is sacrificed to check the invariants. However, due to the dominance of functional and object oriented programming methodologies, having functions, classes, and methods is inevitable. Thus, in the next section, we investigate where to insert invariant checks

for maximum gain and minimum performance degradation.

5.2 Insertion Locations for Invariant Checks

If we have invariants obtained at the level of functional granularity, they are relevant in two places: at the entrance point of the function and at its exit point(s). Therefore, checks can be inserted for them either outside of the function, in the caller's code, or inside the function, in the callee's code. Invariants can be checked at either of these locations since Daikon derives the invariants it suggests from parameters passed to callee. Therefore, these values are accessible in both the calling and the called function. However, performance penalties differ due to the spatial locality of the variables being checked.

Checked outside of function	Checked inside of function
Vector Addition Function: Invariant Checking Inside Function	Vector Addition Function: Invariant Checking Outside of Function.

void foo(int cnt,int *c, int* a,int* b)
{
int i;
//Check before
if (sizeof(a)!=sizeof(b)) i_flag=1;
if (sizeof(a)!=sizeof(c)) i_flag=1;
for (i=0;i<cnt;i++){
c[i]=a[i]+b[i];
}
//Check after
for (i=0;i<cnt;i++){
if (c[i]!=a[i]+b[i]) i_flag=1;
}
}
int main ()
{ …..
foo(100,c,a,b);
…..
}
	void foo(int cnt,int *c, int* a,int* b)
	{
	int i;
	for (i=0;i<cnt;i++){
	c[i]=a[i]+b[i];
	}
	}
	int main ()
	{ …..
	//Check before
	if (sizeof(a)!=sizeof(b)) i_flag=1;
	if (sizeof(a)!=sizeof(c)) i_flag=1;
	foo(100,c,a,b);
	//Check after
	for (j=0;j<100;j++{
	if (c[j]!=a[j]+b[j]) i_flag=1;
	}
	……
	}

Figure 5. Different Locations to Insert Invariants

As an aside, the latter method is made more complex by functions with multiple exit points. However, we consider only functions with a single exit point, as any algorithm can be represented as a proper program (with one entry and one exit point).

Table 1 displays the result of an experiment that inserts invariants in different locations. The test was performed on programs that implemented various vector computations by passing local array pointers to the functions. The results were interesting, as they differed from our expectations. We hypothesized that inserting invariants outside of called function would be faster, since checking the added invariants involves accessing variables allocated on the call stack of the calling function. However, the simulation revealed the opposite behavior. Inserting invariants on the caller's side caused a higher instruction cache miss rate and degraded performance. We presume the results are related to speculation failures, because our invariant checks are implemented with if-statements. Furthermore, we hypothesized that if speculation errors caused the higher rate of instruction cache misses when invariants were inserted outside of called functions, manipulating the branch prediction scheme would reveal that trend.

	Vector addition		Vector Multiply		Vector division	
	Inside	Outside	Inside	Outside	Inside	Outside
Cycle	1464732	1510387	1508036	1520287	1855598	1856132
IL1. miss	1162	4135	1364	4135	1780	1947
DL1. miss	181	191	193	191	311	313
L2 lookup	1343	4326	1557	4326	2093	2262
Branch miss	344	1531	1531	1531	11404	11405

Table 1. Performance comparison between inserting invariant checks outside of the function (in the caller's body) versus inside of function (in the callee's body)

We tried 5 different branch prediction schemes to test this hypothesis. As Figure 6 shows, the type of branch prediction used does not affect the pattern of higher instruction cache misses. All of the variants have higher level 1 instruction cache miss rates when invariants are added inserted outside of functions. Thus, we must conclude that this phenomenon is independent from the branch prediction scheme.

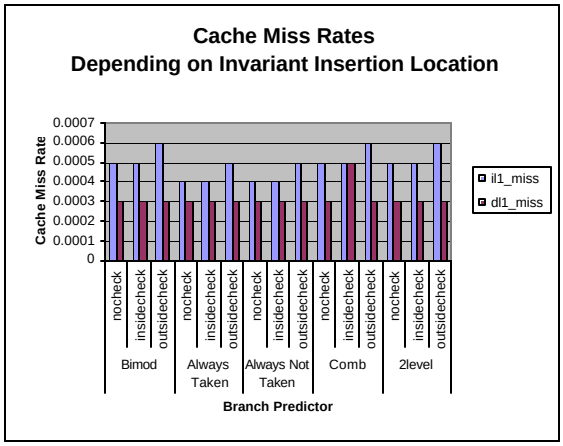

Figure 6. Level 1 Instruction Cache Miss Rate with Different Branch Predictors

Our next hypothesis involved the capacity of the level 1 instruction cache size. Figure 7 reveals the results of our experiments with different level 1 cache sizes. The larger sized

instruction caches have closed the performance gap between inserting invariants inside of the called function and outside of the function. However, while closer together, the pattern still shows up.

# of Cache Lines Level 1 I-Cache	Invariant Check Location	Cycle	Level 1 I- Cache Miss Rate	Level 1 D- Cache Miss Rate
512	No check	1851377	0.0005	0.0003
	Inside	1855598	0.0005	0.0003
	Outside	1856132	0.0006	0.0003
1024	No check	1848966	0.0004	0.0003
	Inside	1853187	0.0004	0.0003
	Outside	1854909	0.0005	0.0003

Figure 7. Level 1 Cache Miss Rates with Different Level 1 Instruction Cache Size

Next, we wondered if we were documenting a phenomenon caused by the limited issue window size in Superscalar processors. As the issue window size became larger, the IPC increased, showing that the amount of available parallelism contributes to reducing the performance gap between different insertion locations. However, this is also not a final solution.

n- way superscalar	Insertion Location	Cycle	sim_IPC	Il1_miss
2way	No checks	2459332	1.3285	0.0005
	Inside	2465138	1.3292	0.0005
	Outside	2464886	1.3283	0.0006
4way	No checks	1851377	1.7648	0.0005
	Inside	1855598	1.7658	0.0005
	Outside	1856132	1.7639	0.0006
8way	No checks	1830714	1.7847	0.0005
	Inside	1834814	1.7858	0.0005
	Outside	1835909	1.7834	0.0006

Figure 8. Performance Difference between Different Issue Windows in Superscalar

Finally, we decided to try the same experiment with a different cache replacement algorithm. As Figure 9 shows, the cache replacement algorithm also doesn't significantly affect this situation. Hence, having reduced most of the major influences on processor performance, we speculate that this performance difference is caused by compiler optimizations and independent of architectural constraints.

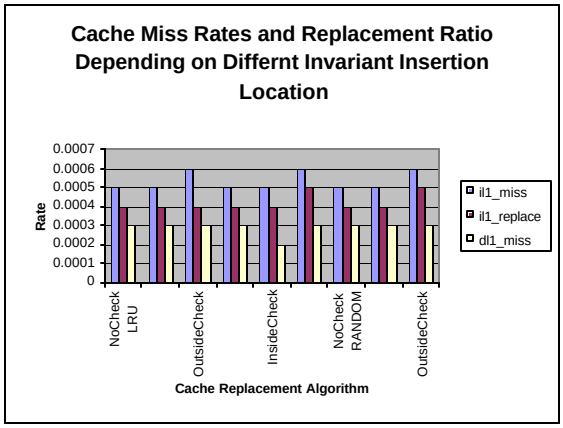

Figure 9. Cache Miss Rates and Replacement Rates with Different Replacement Algorithms

5.3 Complexity of Invariant Checks

We have identified three major types of complexity when dealing with invariants: spatial, which refers to the need for extra variables; temporal, which describes the varying numbers of operations that a single invariant might include; and functional, which refers to the necessity of calling library functions in invariant checks.

5.3.1 Spatial Complexity

When checking invariants, state must sometimes be stored, since some variables are overwritten during execution of a function. New variables of the same type and size must be allocated to store the old state.

```
Function accumulation(array A, array B) {
int s= sizeof (A);
for (int I=0;I<s;I++) {
  A[I]=A[I]+B[I];
}
```

When we want to check the invariant "post_A[I] ==pre_A[I]+pre_B[I]" in the code above, we need to store pre_A into a temporary array T. Copying the values in array A to T adds O(sizeof(A)) bytes of spatial complexity to the system. Table 2, below, compares the computation times for adding new storage to check invariants, rather than using memory that is already existant in the program. On our test programs, adding new storage to check the invariants took, on average, 11% more

computation time, even the same invariants are checked.

	Operation	# Instruction	# cycle
Needs Additional storage	A[k]=A[k]+B[k]	759602	353224
Needs No additional storage	C[k]=A[k]+B[k]	660335	317134

Table 2. Performance comparison between invariant checks that require additional data storage and those that do not.

5.3.2 Temporal Complexity

Checking invariants for large data types (like arrays or trees) can be computationally intensive. For example, checking invariants for every element of a single-dimensional array takes O(sizeof(array)) time. By checking a randomly selected subset of the array, only O(c) time is consumed, where c is a predetermined value, and there is some net gain in our confidence in the computed values. However, this is balanced by possibility of missing more computation failures. A check on random pieces of the data structure will catch an error that occurs in most (or all) elements, but it will probably not catch an error that involves only a single element. Nevertheless, the simulations results show that we can save at least 10% of total execution time by only checking invariants for some randomly selected subset. More simulation results are needed for different data structures and programs, and we must determine how to best balance computation confidence and performance cost.

5.3.3 Functional Complexity

Inserting invariants that verify library function calls is difficult, and some programs make huge amounts of calls to library functions. Daikon cannot detect invariants that involve library functions, so the insertion must be done manually. Also, despite knowing the checks inside of function calls are faster, library functions cannot be modified, so invariant checks must be added outside of the called function. Many of the checks added involved either checking that the return value was in an appropriate range or simply re-executing the library call and comparing the two return values. For example, for cos(x) and sin(x), we can insert weak invariants such as cos(x) and sin(x) are in the range –1 to 1. However, for tan(x), we had to re-compute tan(x) and compare the result with the previous outcome to increase confidence, since the range of tan(x) is not bounded. Re-executing library calls can be prohibitively expensive, and the methods required to check library calls are distinctly different from the time complexity mentioned in Section 5.3, so we classify these checks as O(f) complexity, with f meaning "function."

6 Analysis

In section 5, we learned that invariant checking has its own temporal and spatial complexity. We also found that inserting invariants inside a function is better than inserting the same checks outside of the function. These are basic properties that can affect how efficient invariant checks can be. Now, we want to investigate the trade-off between performance sacrifice and confidence gain using several realistic benchmark programs.

In section 6.1.1, we use linked lists and binary search tree programs as small examples of realistic, pointer-heavy applications. We believe these programs are characteristic of general applications, which often use large amounts of pointers and recursion. Next, in section 6.1.2, we use mathematical analysis programs to model scientific applications that require a high level of accuracy. Finally, in section 6.1.3, we display our results for one integer and one floating-point application from the Spec 2000 benchmark suite.

6.1 Benchmarks

Thus far, our investigation has gathered data using overly simplified toy programs with very limited characteristics. We wanted to see how much performance is sacrificed on real systems that employ invariant checking.

6.1.1 General applications

For models of general applications, we chose several programs that use common abstract data types, including one that features a binary search tree and one that utilizes a linked list. These programs were more challenging than the "toy" programs we tested in earlier sections in three important ways: heavy pointer use, complex invariants, and recursive structure.

First, both of these programs feature extensive use of pointers. This presents a large challenge to invariant checkers, and Daikon is no exception. It had difficulty obtaining interesting invariants for these programs, as it is not good at capturing relationships between dereferenced

pointers. Second, the two data structures feature only fairly complex, difficult to check invariants, which we believe is characteristic of many general applications. However, lists and trees are well studied in the literature, so we manually added invariants that check that the basic properties of the data structure are maintained. (For example, one invariant of a list is that after insertion, the size of the list increases by no more than one.) Finally, the recursive nature of lists and trees, in addition to the explicit recursion used in the implementation of the binary search tree, adds complexity to the problem. To check many invariants (for example, that the tree is not altered during a "find" operation), every node must be revisited, using a recursive function with (possibly) heavy overhead.

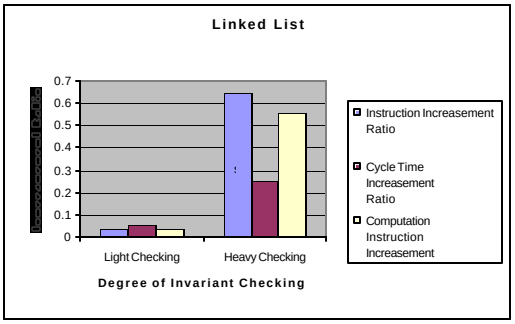

Figure 10. Linked List: Performance Sacrifice and Coverage

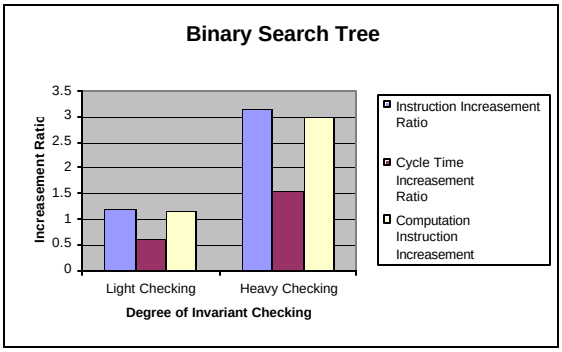

Figure 11. Binary Search Tree: Performance Sacrifice and Coverage

The combination of these three factors led to the results we see below. While we were able to significantly increase our confidence in the correctness of the program's execution, this result was won with an extremely high performance sacrifice. Consider a simple find function, which does not need to visit every node in the list or tree to execute properly. As part of our effort toward verifying that this function worked correctly, we can check that the size of the structure does not change. To check this, however, the invariant check must visit every node, which makes the invariant check significantly more expensive than the function it is checking. In this case, we believe that it is definitely more worthwhile to either scale down the number (or complexity) of the invariants checked (which reduces our confidence in the result) or to simply re-execute the program

Figure 10 and Figure 11 shows that as we add more invariant checks, we have a higher *Computation Instruction Increase* with a lower *Cycle Time Increase Ratio.*

6.1.2 Scientific applications

We believe our verification technique is especially applicable for scientific computing problems. These problems could benefit from the added confidence in the results that invariant checking brings, and invariants are discovered easily for such programs, since they are often based directly on mathematical relationships that can be checked. We prepared several mathematical analysis programs for these simulations, including applications using Taylor series, Machine Epsilon, Newton's method, polynomial interpolation, Spline approximation.

These mathematical analysis programs all used library functions from "math.h". Daikon ignores library function when detecting invariants, so using our invariant checking technique required that we re-execute the library functions. Since these invariant checks are expensive and distinctly different from the time complexity mentioned in Section 4.3, we call this O(f) complexity, with f standing for "function."

For each program, we added invariants that can be checked without calling library functions again. Next, we added O(f) complexity invariants. For example, tan(x) is not bounded by any value, so to increase our confidence level in the correct computation of tan(x), we recomputed tan(x) and compared it with previously computed value.

We found interesting results throughout these experiments. Even though we added more invariants, the performance sacrifice from invariant checking we documented was not as expensive as the cost of executing the instructions we covered. Also, we noticed that the cycle time sacrifice is not always in a 1:1 correspondence with the number of invariants added.

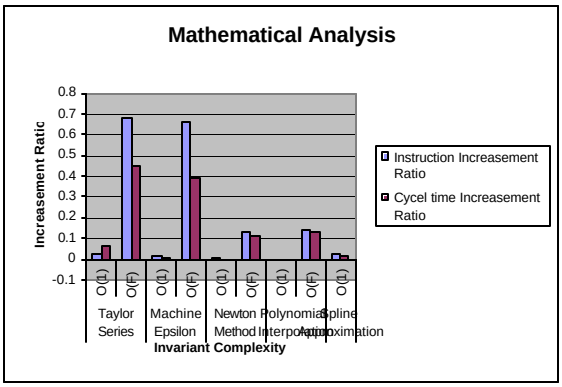

Figure 12. Mathematical Analysis Programs: Performance Sacrifice and Coverage

6.1.3 Spec 2000

While the previous two sections have exercised our invariant insertion technique on more realistic programs, we still needed a basis upon which to compare our work to other techniques. Therefore, we chose to instrument two benchmarks from the Spec2000 CPU Benchmark suite that exhibited extremely different program characteristics: art, from the floating point suite, and mcf, an integer program.

It should be noted that since we ran these programs on SimpleScalar simulator, we were *not* able to run these benchmarks in an official, reportable manner. The runspec utility could not be used to execute the instrumented executables, so we cannot guarantee that we used the official command line options, although every effort was made to ensure that the programs were invoked correctly. In addition, in the interest of time, instead of using the full test suites, the "test" test suites were used. [9]

6.1.3.1 179.art

art is an image recognition program that trains a neural net using a series of training images and then attempts to recognize and categorize images from a separate set of images. We chose art from the floating point suite since it features a large number of inner loops, uses a large amount of memory, is CPU intensive, and relies heavily on array operations, including multiplication. In this sense, it resembles many of the toy programs we used earlier, but it dwarfs them in terms of scale.

Invariants were inserted to art in two levels to get some idea of the relation between increasing confidence and decreasing performance. The "light" level contains roughly half as many checks as the "heavily" instrumented program. However, in both cases, no additional variables were added, and only invariants of O(1) temporal complexity were inserted. (See sections 5.3.1 and 5.3.2 for a more complete discussion on temporal and spatial complexity.) Therefore, the invariants inserted were simple and checked single values, array boundaries, and counters. We did not insert O(n) invariants (which would have allowed us to verify that all of the array operations were completed correctly) so that we could more easily compare our data for art with the data obtained for mcf, which does not provide many opportunities for O(n) complexity checks.

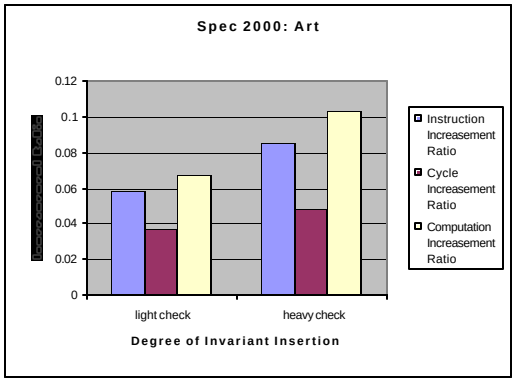

Figure 13. Spec2000 Art: Performance Sacrifice and Coverage

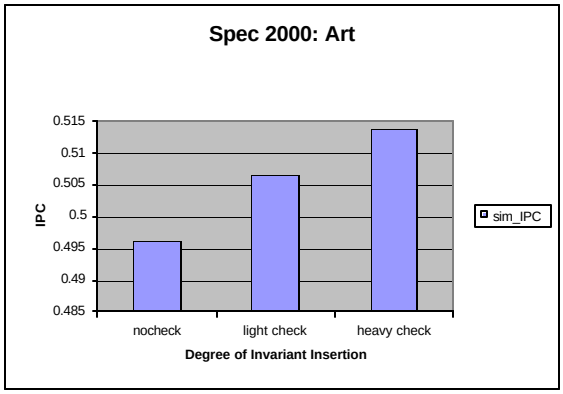

Figure 14. Spec2000 Art: IPC

Figure 13 and Figure 14 describe the results we obtained from running SimpleScalar on art and our instrumented variants of art. The IPC increases as more invariants are added, which suggests that even more checks could have been inserted while continuing to take advantage of empty issue slots. It is especially nice that the *computation increase ratio* grows nearly six times as fast as the *cycle increase ratio*, although it is too early to declare a trend, as we have only two points.

6.1.3.2 181.mcf

mcf was derived from a resource scheduling program: specifically, mcf was originally a "single-depot vehicle scheduler." [9] It almost exclusively utilizes integer arithmetic and features a huge amount of pointer swapping and dereferencing. As such, it is not overly intensive, computationally, and poses a real problem for Daikon, which does not handle pointers well. In this sense, it is reminiscent of the "general application" programs we tested in section 4.4.1, with one key difference: mcf does not utilize highly recursive structures.

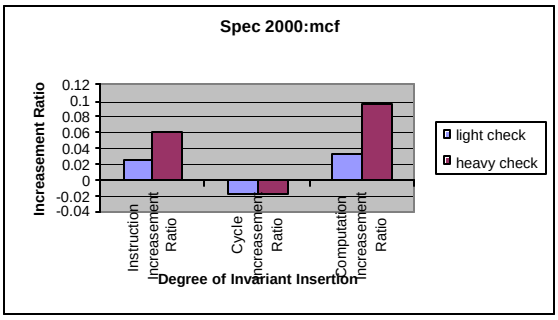

Figure 15. Spec2000 Mcf: Performance Sacrifice and Coverage

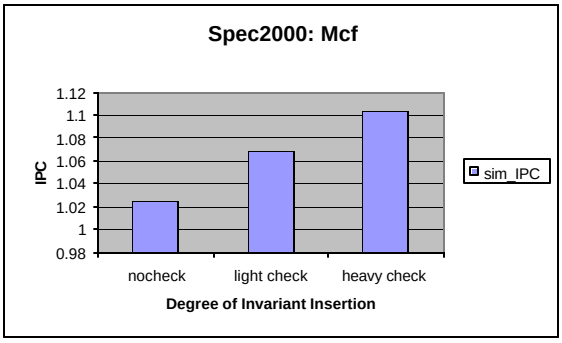

Figure 16. Spec2000 Mcf: IPC

Like in art, invariants were added to mcf in two levels, with the "heavy" level of invariant checking having about twice as many checks as the "light" level. Again, no extra variables were added to the program (with the exception of the global error flag), and only O(1) complexity invariants were considered. Most of those used in to instrument mcf consisted of NULL pointer checks, verification that a single value lay within a legal range, and array boundary checks.

Figures 15 and 16 describe the data that we obtained from simulating the original mcf and our instrumented versions. The IPC increases even more dramatically than it does in art, and we received some extremely strange results for the *instruction increase ratio* and the *computation increase ratio*: in some cases the ratios are negative, indicating a reduction from the original. We have tried isolating the reason for these results but, at the moment, can only conclude that the compiler was able to optimize the code more effectively or that SimpleScalar ran into simulation problem.

6.2 Result

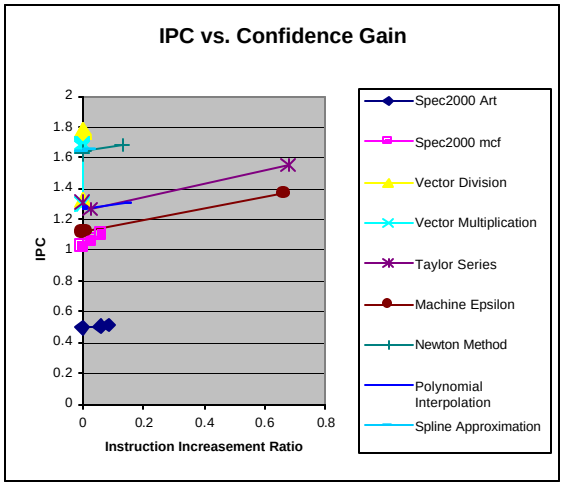

Figure 17. IPC vs. Confidence Gain: IPC

Figure 17, above, shows that in each program, adding more invariant checks generally increased the IPC. We believe that invariant checks often use data that is already resident in the cache, so they can be executed without memory latency, which increases IPC. Hence, we think invariant checking can use the empty issue slots in a superscalar processor extremely efficiently.

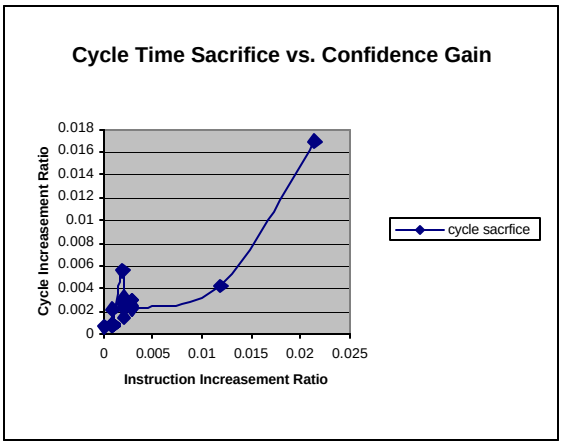

Figure 18. Cycle Time Sacrifice vs. Confidence Gain

Figure 18 shows that cycle time sacrificed increases, in general, more slowly than confidence gain, which implies that adding invariant checks is more efficient than re-execution of the code.

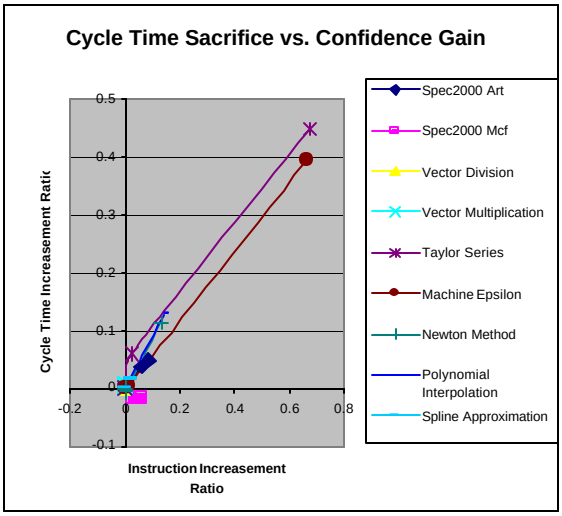

Figure 19. Cycle Time Sacrifice vs. Confidence Gain

From the graph of Cycle Time Sacrifice vs. Confidence Gain (Figure 19), we saw that our two metrics are correlated in a linear relationship, so the performance sacrifice is always roughly the same as the confidence gain.

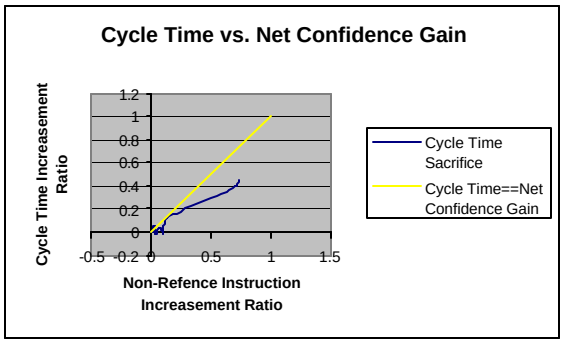

Figure 20. Cycle Time vs. Net Confidence Gain: Net Confidence Gain is approximated using the Non-Reference Instruction Increase heuristic

Figure 20 shows the relationship between Cycle Time and Net Confidence Gain. The gap between the break-even point and the slope of *Computation Confidence Gain* is even larger than that of the *Confidence Gain*. This means with the same performance sacrifice, we get a more effective verification, because with the same cost, we contribute more towards non-memory computations. We think the more invariant checks that are inserted, the better the cache hit rate, and therefore, memory latency is, relatively speaking, less of a constraint on parallelism.

7 Related Work

Traditionally, verification of programs has lain entirely in the domain of software engineering (especially in the field of type safety). As reliability in complex systems becomes a larger concern, due to the ever-increasing cost of verifying design correctness and the growing possibility of faults caused by interference (natural or otherwise), fault tolerant architectures are attracting more interest in the architectural community. Much of the work done to date has focused on attempting to guarantee correctness of execution with respect to the code being run, usually by adding a secondary processor, rather than increasing confidence in the program's correct execution, as we have done, which is reminiscent of the work done in software engineering and type safety.

We based our work on an idea by Jeong and Jamison [6], which was, in turn, an extension of Necula's idea of proof-carrying code [7] and an alternative to the DIVA [1] architecture. Proof-carrying code includes a proof of safety that guarantees safe execution if the accompanying

invariants hold. Similarly, in Jeong and Jamison's work, programs are annotated with invariants that, while not guaranteeing proper execution, increase confidence in the executed instructions if they hold. They use a dual processor system (with a fast primary processor and slower secondary processor) that simultaneously executes the primary program (on the faster processor) and verifies groups of instructions (on the slower processor) by computing the invariants inserted by the programmer. If an invariant is violated, the corresponding block of code is re-executed.

While Jeong and Jamison verify code at the "block" granularity, DIVA verifies (through re-execution) every instruction, offers a stronger guarantee of execution correctness with respect to the provided program, and greatly increases the fault tolerance of the system. Austin's goal in this research was to reduce hardware design costs and the need for completely correct processors by introducing a second processor that recomputes all of the results of the primary processor. The second processor is much simpler than the first, as it does not have to perform branch or data speculation since the first processor has already done that work. In addition, it is slower. This combination of factors makes the second processor less prone to noise-related faults and easier to design and verify. However, the DIVA architecture does include additional hardware, and it must rely on the correctness of the program it is executing, while our system can, with annotations provided by the programmer, catch errors in the code itself.

In contrast to DIVA, which focuses on verifying correctness at the hardware level, Chong [4] introduced hardware structures that support execution verification at the software level. For example, he is interested in a hardware access table (HAT) to accelerate table lookups that are critical for locating memory access and concurrency errors. (Austin's [1] spatial and temporal memory access checking could take advantage of this, for example.) Chong's efforts are more directed towards error detection in programs during software development (rather than verification of processor execution in general), and while the hardware support that he espouses provides greatly decreases the performance penalty associated with dynamic pointer checking, it does not, by itself, increase the ruggedness of the system.

8 Conclusion

We have presented data showing that adding invariants to programs to verify that they are correctly computed is feasible, in terms of performance, especially since invariant checks tend to make use of empty issue slots and increase parallelism. Furthermore, we have done small studies that indicate that the best way to insert such checks is to focus at a functional granularity, to place the invariants inside the called function, and to perform only the simple, O(1) checks. To make these assertions, we developed a framework for measurement that compares performance sacrifices (or computational throughput) to the amount of confidence that the inserted checks give us.

Nevertheless, this is only a tiny amount of the work that must be done before this technique can truly be deemed worthwhile. In the future, we would like to focus upon determining the true effectiveness of the inserted checks and gathering more varied data points from which to draw conclusions. To start, we would like to simulate errors in computation by modifying SimpleScalar to randomly insert errors in its simulation at varying frequencies, and then we would like to complete our instrumentation of the Spec2000 benchmarks. Finally, even if this technique continues to show performance, huge advances in invariant generation technology will have to be made. The current generation of technology is simply not at the point where invariant checks can be automatically inserted into code by a compiler or some other tool.

Overall, we believe that the future technological advances will lead to an unavoidable increase in computation errors committed by processors. Techniques for increasing the fault tolerance of systems must be developed to deal with this trend as well as to allow us to build extremely fast systems that are not limited by our current fixation on absolutely correct computation.

Acknowledgements

We would like to thank Tim James and Mark Oskin for all of their guidance and the huge amount of technical support they provided during the quarter. Thanks also to Michael Ernst, for all of his work on Daikon.

References

[1] Todd M. Austin, "DIVA: A Dynamic Approach to Microprocessor Verification," in the *Journal of Instruction-Level Parallelism*, May 2000.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[3] Doug Burder and Todd M. Austin, The SimpleScalar Tool Set, Version 3.0.

[4] Frederick Chong et al, "Hardware Support for Software Safety," 2002.

[5] Michael D. Ernst, "Dynamically Detecting Likely Program Invariants," PhD Dissertation, University of Washington, August 2000.

[6] Jaein Jeong and Jonathon Jamison, "Verifying Architecture," 2000.

[7] George C. Necula, "Proof-Carrying Code," in *Proceedings of the Principles of Programming Languages (POPL)*, January 1997.

[8] Jeremy W. Nimmer and Michael D. Ernst, "Invariant Inference for Static Checking: An Empirical Evaluation," ACM, 2001.

[9] Standard Performance Evaluation Corporation, http://www.specbench.org/osg/cpu2000/, 2000.