International Journal of Sciences: Basic and Applied Research (IJSBAR)

ISSN 2307-4531
(Print & Online)

http://gssrr.org/index.php?journal=JournalOfBasicAndApplied

SAGE: Profiling and Optimization in Formal Methods

Sandhya P N[a]*, Vidya H A[b]

[a]*Assistant Professor, Dept. of CSE.,CIT,Gubbi,Tumkur-572216,Karnataka,India*

[b]*Assistant Professor, Dept. of CSE.,CIT,Gubbi,Tumkur-572216,Karnataka,India*

[a]*Email: sandhya.pn@cittumkur.org*

[b]*Email: vidya.ha@gmail.com*

Abstract

The quality and the correctness of software are often the greatest concern in electronic systems. Formal methods techniques and tools provide a guarantee that a design of software system is free of specific flaws. This paper discusses an open source mathematical software system SAGE, which combines and extends program analysis, testing, verification, model checking, and automated theorem proving techniques. Then we concentrate on profiling and optimization techniques in analysis and design phase of software systems in SAGE. In today's programming world optimization plays a vital role. It is an on-going, non-functional requirement that affects all stages in the development of a system, from analysis and design through development and implementation. Profiling refers to analysis of the relative execution time spent in different parts of the program. Finally, we describe various optimization techniques.

Keywords–SAGE; profiling.

1. Introduction

"Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase "mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic.

---

* Corresponding author.

E-mail address: sandhya.pn@cittumkur.org.

The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design (whether hardware or software) and establish a correctness or safety property that is true for all possible inputs [1].

The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design[2].Formal method demonstrates the absence of undesirable system behavior.

Formal methods have offered great benefits, but often at a heavy price. For everyday software development, in which the pressures of the market don't allow full-scale formal methods to be applied, a more lightweight approach is called for. A lightweight approach emphasizes partiality and focused application, can bring greater benefits at reduced cost[3].

1.1 Analyzing and Proving Properties of Systems and Specifications

Once a formal specification has been constructed, it is possible to analyze, manipulate, and reason about it just like any other mathematical expression. A significant difference between formal specifications of software and the more familiar mathematical expressions of algebra or calculus, however, is that specifications are typically much larger - often hundreds or thousands of lines. To deal with the size and complexity of these expressions, many varieties of software tools have been constructed. These tools can be grouped broadly into two categories: theorem proving tools and model checkers.

Theorem proving tools assist the user in constructing proofs, generally to show that the specification has desired properties such as absence of deadlock or various security properties. These tools require a degree of skill to use, but they can handle very large specifications with complex properties. A more recent development in formal methods has been the introduction of model checkers, which explore enormous state spaces that cover, to some degree, all possible executions of the specified program. They can demonstrate that the specified program has a desired property by exploring all possible executions, or produce counterexamples where the property does not hold. Although these tools can be made fully automatic, they cannot handle problems as large or varied as theorem proving tools. The most sophisticated tools combine aspects of both model checkers and theorem provers, doing model checking on some parts of the specification, but relying on user guidance to prove difficult properties. This section shows the types of tools that are available, the tradeoffs involved, and appropriate problem domains for the different tools[4].

Theorem proving tools have been developed to aid the human in demonstrating that a program meets its specification. There are many theorem proving tools available including: PVS, ACL2, HOL, Isabelle, Nuprl, Z/Eves, SCR, and the B-Tool (See [14] for a fuller list.)

1.2 Model Checkers

One of the challenges faced by formal methods researchers is the need to make formal techniques available to the broadest possible audience of developers. Model checkers attempt to make formal techniques easier to use by providing a high degree of automation at the expense of generality. Inputs to a model checker are typically a finite state model of a system, along with a set of properties that are expected to be maintained by the system. Properties to be verified can usually be categorized as one of the following[5]:

1. Correct sequences of events

2. Proper consequences of activities

3. Simultaneous occurrence of particular events

4. Mutual exclusion of particular events

5. Required precedence of activities

The model checker effectively explores all possible event sequences of the finite state model to determine if properties, typically expressed in temporal logic, always hold. Because the model is finite, the state space search is guaranteed to terminate. If properties hold, the model checker outputs a confirmation. If a property fails to hold for some possible event sequences, the tool produces counterexamples, i.e., traces of event sequences that lead to failure of the property in question. A useful trick that can be applied in model checking is to specify that the negation of a desired property that should hold. In this case, the model checker produces an event trace that results in a failure of the negated property. This event sequence, therefore, is actually a valid trace of the system, which can be post processed to generate complete test cases (i.e., both test input and expected system result) [6].

The two greatest advantages of model checkers are their fully automatic analysis and their ability to produce counterexamples that can be used in testing or other analysis. Their disadvantage results from the tradeoffs made to make automation possible. Because they effectively explore a complete state space, the complexity of analysis normally grows exponentially with the size of the model. To reduce the size of the finite model, abstractions must be used, such as reducing the range of discrete variables, or using a simple predicate to represent a more complex condition. As a result, model checkers can only be used to analyze an abstraction of the system, rather than a complete model, so they are sometimes regarded as debugging tools (for specifications) rather than verification tools [15]. Despite their limitations, model checkers have been applied to significant real-world systems, particularly integrated circuit design and cryptographic protocols. Abstraction techniques make it possible to model systems with 200 variables, and up to 10^{120} reachable states [7]. Table I summarizes differences between model checkers and theorem proving tools.

Table1. Summary of model checker and theorem prover properties

Model checkers		Theorem provers
Typical notation	FSMs, temporal logic	First order, higher order logics
Method of operation	Automatic	Semi-automatic
Output	Confirmation or counterexample	Proof (if successful)
Range of applicability	Bounded, finite models	Essentially unlimited

One of the most interesting applications of formal methods has been the development of tools that can generate complete test cases from formal specifications.

The expected system response can only be determined by reading the specification; programmers are expected to provide this critical missing link in most automated testing systems. The great advantage of test generation tools based on formal methods is that a formal specification describes system behavior mathematically, so expected system responses for particular inputs can be generated; i.e., the tool can generate complete test cases, not just test data input or test scaffolding.

1.3 Advantages of formal methods

From a cost-benefit standpoint, generating tests from specifications can be one of the most productive uses of formal methods. Approximately half the staff time in a typical commercial software development effort is spent on testing. As computer users have recognized, even this level of effort only removes the most obvious flaws. Much of the software industry operates under a marketing strategy that gives feature richness and time to market a higher priority than quality, because users have demonstrated a willingness to accept bugs in return for more features. Some of the newer test generation tools hold the promise of improving quality while simultaneously reducing time to market, because less developer time is spent on programming test cases. These tools can also provide benefits for custom software, such as most M&S systems, by reducing the time spent on test development and thus allowing more time for other tasks. Some empirical measurements have shown that tests generated by these tools provide test coverage as good or better than that achieved by manually generated tests, so developers can choose between producing more tests in the same number of staff hours, or reducing the number of hours required for testing.

The formal analysis helps to prevent unexpected failures that can occur in large distributed systems where processes occur with partial human intervention. By modeling processes at the requirements stage, developers can identify problems that might require major rework if not detected until the system is built and tested.

2. What is SAGE?

SAGE (*scalable, automated, guided execution*) is a free open-source mathematics software system licensed under the GNU Public License (GPL). It combines the power of about 100 open-source packages with a large amount of new code to provide a free open source platform for mathematical computation. Sage has many notable features.

- Sage is free, due mainly to the volunteer effort of hundreds of people and generous funding from the National Science Foundation, private donations, and other organizations such as Google and Microsoft. There are no license codes or copy protection. Sage is also open source, so there are absolutely no secret or proprietary algorithms anywhere in Sage. There is nothing that you are not allowed to see or change.
- Sage uses the mainstream programming language Python. Learning Sage will make you proficient in this popular, widely used, and well supported free programming language, which you will likely also use for other non-mathematics projects. Moreover, Sage features the Cython compiler, which allows one to combine Python, C/C++/Fortran libraries, and native machine types for potentially huge speedups.
- Sage is uniquely able to combine functionality from dozens of other mathematical software programs and programming languages via smart pseudo terminal interfaces. You can combine Lisp, Mathematica, and C code to attack a single problem.
- Sage has both a sophisticated multiuser web-based graphical user interface and a powerful command line interface. Sage can also be made to work with any other Python interactive development environment (IDE).
- Sage may have the widest range of mathematical capabilities of any single mathematical software system available. Sage and its components are developed by an active and enthusiastic worldwide community of people from many areas of mathematics, science, and engineering.
- Modifications to Sage are publicly peer reviewed, and what goes into Sage is decided via community discussions; no matter who you are, if you have a brilliant idea, the energy, and can clearly argue that something should go into Sage, it probably will. Known bugs in Sage, and all discussions about them are available for all to see.

Sage is nothing like Magma, Maple, Mathematica, and Matlab, in which details of their implementations of algorithms is secret, their list of bugs is concealed, how they decided what got included in each release is under wraps, their custom programming language locks you in, and you must fight with license codes, copy protection and intentionally crippled web interfaces[8].

3. SAGE Architecture

The high-level architecture of SAGE is depicted in fig.1.Given an initial input, Input0, SAGE starts by running the program under test with AppVerifier to see if this initial input triggers a bug. If not, SAGE then collects the list of unique program instructions executed during this run. Next, SAGE symbolically executes the program

with that input and generates a path constraint, characterizing the current program execution with a conjunction of input constraints.

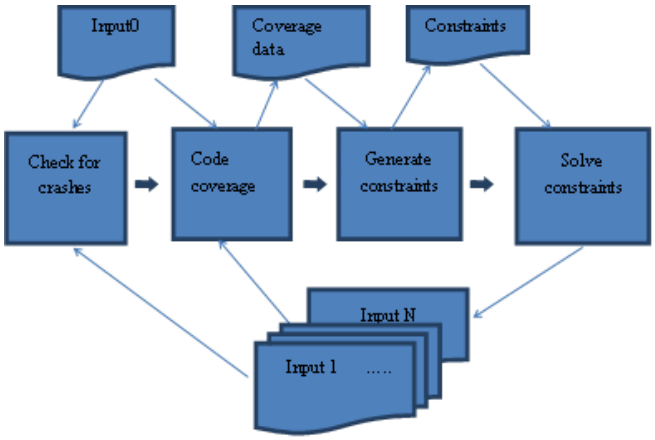

Fig. 1. SAGE Architecture

Then, implementing a generational search, all the constraints in that path constraint are negated one-by-one, placed in a conjunction with the prefix of the path constraint leading to it, and attempted to be solved by a constraint solver (we currently use the Z3 SMT solver [8]). All satisfiable constraints are then mapped to N new inputs. These N new inputs are then tested and ranked according to incremental instruction coverage. For instance, if executing the program with new Input1 discovers100 new instructions, Input1 gets a score of 100, and so on.

Next, the new input with the highest score is selected to go through the (expensive) symbolic execution task, and the cycle is repeated, possibly forever. Note that all the SAGE tasks can be executed in parallel on a multi-core machine or on a set of machines[9].

4. Profiling

Sometimes it is useful to check for bottlenecks in code to understand which parts take the most computational time; this can give a good idea of which parts to optimize. Cython and therefore Sage offer profiling technique.

4.1The need for profiling

Since it's common that programs spend most of the execution time in a few small parts of the code (usually some loops), it is recommended to perform first "profiling", i.e. analysis of the relative execution time spent in different parts of the program.

Profiling is also important when you apply optimizations, compilers and computer hardware are complicated systems, and the results of applying an optimization may be hard to guess.

The simplest to use is the prun command in the interactive shell. It returns a summary describing which functions took how much computational time. To profile (the currently slow! - as of SAGE version 1.0) matrix multiplication over finite fields, for example, does:

The pure C code: test.c

```
int mat_mul(int A[][], int A[][]) {
for(i=0;i<a_r; i++)
{
      for(j=0;j<a_c; j++)
      {
            for(k=0;k<b_c; k++)
            { B[i][j]=B[i][j]+A[i][k]*A[k][j];}
      }
}
}
```

The Cython code: test.spyx:

```
cdef extern from "test.c":
int mat_mul(int A[][], int A[][])
def test(A,A):
      return mat_mul(A,A)
```

Then the following works:

```
sage: attach "test.spyx"
Compiling (...)/test.spyx...
sage: %prun B = A*A
      32893 function calls in 1.100 CPU seconds

Ordered by: internal time
ncalls tottime percall cumtime percall filename:lineno(function)
12127  0.160    0.000    0.160  0.000    :0(isinstance)
 2000  0.150    0.000    0.280  0.000    matrix.py:2235(__getitem__)
 1000  0.120    0.000    0.370  0.000    finite_field_element.py:392(__mul__)
 1903  0.120    0.000    0.200  0.000    finite_field_element.py:47(__init__)
 1900  0.090    0.000    0.220  0.000    finite_field_element.py:376(__compat)
  900  0.080    0.000    0.260  0.000    finite_field_element.py:380(__add__)
    1  0.070    0.070    1.100  1.100    matrix.py:864(__mul__)
 2105  0.070    0.000    0.070  0.000    matrix.py:282(ncols)
 ...
```

Fig. 2. Running prun command in sage for matrix multiplication to get profiled data

Here in fig. 2 n calls is the number of calls, to t time is the total time spent in the given function (and excluding time made in calls to sub-functions), percall is the quotient of to t time divided by n calls. Cum time is the total time spent in this and all sub-functions (i.e., from invocation until exit), per call is the quotient of cum time divided by primitive calls, and file name: line no (function) provides the respective data of each function. The rule of thumb here is: The higher the function in that listing, the more expensive it is. Thus it is more interesting for optimization.

The profiling data may be written to an object as well to allow closer examination:

```
sage: %prun -r A*A

sage: stats = _

sage: stats?
```

Note: entering`stats = prun -r A\*`displays a syntax error message because prun is an IPython shell command, not a regular function.[8]

5. Optimization Techniques For Sage

Optimization is an on-going, non-functional requirement that affects all stages in the development of a system, from analysis and design through development and implementation. The analysis phase will identify the areas that need to be addressed by optimization, the design phase will indicate how major optimization should be achieved, and the development phase will apply these optimizations. During the implementation phase while the system is deployed on the appropriate hardware resources, further areas of optimization will be identified by performance monitoring during application testing. The example in this section (fig.2) shows the total time taken for executing the loops.

As time is an important constraint we can reduce the time of evaluating the loops by applying some optimization techniques in the design phase. The following are some of the optimization techniques for loops that can be applied.

5.1 Loop nest optimization

In computer science and in particular compiler design, Loop nest optimization(LNO) is an optimization technique that applies a set of loop transformations for the purpose of locality optimization or parallelization or other loop overhead reduction of the loop nests. One classical usage is to reduce memory access latency or the cache bandwidth necessary due to cache reuse for some common linear algebra algorithms.

Example: Matrix multiply

Many large mathematical operations on computers end up spending much of their time doing matrix multiplication. The operation is:

$$B = A \times A$$

Where A and B are N×N arrays. Subscripts, for the following description, are in the form B[row][column].

The basic loop is:

```
int i, j, k;
for(i=0;i<N;++i)
{
    for(j=0;j<N;++j)
    {
        B[i][j]=0;
        for(k=0;k<N;++k)
         B[i][j]+=A[i][k]*A[k][j];
    }
}
```

There are three problems to solve:

1. Floating point additions take some number of cycles to complete. In order to keep an adder with multiple cycle latency busy, the code must update multiple accumulators in parallel.
2. Machines can typically do just one memory operation per multiply–add, so values loaded must be reused at least twice.
3. Typical PC memory systems can only sustain one 8-byte doubleword per 10–30 double-precision multiply–adds, so values loaded into the cache must be reused many times.

The original loop calculates the result for one entry in the result matrix at a time. By calculating a small block of entries simultaneously, the following loop reuses each loaded value twice, so that the inner loop has four loads and four multiply–adds, thus solving problem #2. By carrying four accumulators simultaneously, this code can keep a single floating point adder with a latency of 4 busy nearly all the time (problem #1). However, the code does not address the third problem. (Nor does it address the cleanup work necessary when N is odd. Such details will be left out of the following discussion.)

```
for(i=0;i<N;i+=2)
{
    for(j=0;j<N;j+=2)
    {
        acc00=acc01=acc10=acc11=0;
        for(k=0;k<N;k++)
        {
acc00+=A[k][j+0]*A[i+0][k];
        acc01+=A[k][j+1]*A[i+0][k];
        acc10+=A[k][j+0]*A[i+1][k];
        acc11+=A[k][j+1]*A[i+1][k];
        }
        B[i+0][j+0]=acc00;
        B[i+0][j+1]=acc01;
        B[i+1][j+0]=acc10;
        B[i+1][j+1]=acc11;
    }
}
```

This code has had both thei and j iterations blocked by a factor of two, and had both the resulting two-iteration inner loops completely unrolled.

This code would run quite acceptably on a Cray Y-MP (built in the early 1980s), which can sustain 0.8 multiply–adds per memory operation to main memory. A machine like a 2.8 GHz Pentium 4, built in 2003, has slightly less memory bandwidth and vastly better floating point, so that it can sustain 16.5 multiply–adds per memory operation. As a result, the code above will run slower on the 2.8 GHz Pentium 4 than on the 166 MHz Y-MP!

A machine with a longer floating-point addslatency or with multiple adders would require more accumulators to run in parallel. It is easy to change the loop above to compute a 3x3 block instead of a 2x2 block, but the resulting code is not always faster. The loop requires registers to hold both the accumulators and the loaded and

reused A and B values. A 2x2 block requires 7 registers. A 3x3 block requires 13, which will not work on a machine with just 8 floating point registers in the ISA. If the CPU does not have enough registers, the compiler will schedule extra loads and stores to spill the registers into stack slots, which will make the loop run slower than a smaller blocked loop.

Matrix multiplication is like many other codes in that it can be limited by memory bandwidth, and that more registers can help the compiler and programmer reduce the need for memory bandwidth. This *register pressure* is why vendors of RISC CPUs, who intended to build machines more parallel than the general purpose x86 and 68000 CPUs, adopted 32-entry floating-point register files.

The code above does not use the cache very well. During the calculation of a horizontal stripe of B results, one horizontal stripe of A is loaded and the entire matrix A is loaded. For the entire calculation, B is stored once (that's good), A is loaded into the cache once (assuming a stripe of A fits in the cache with a stripe of A), but A is loaded N/ib times, where ib is the size of the strip in the B matrix, for a total of N^3/ib doubleword loads from main memory. In the code above, *ib* is 2.

The next step to reducing the memory traffic is to make ib as large as possible. We want it to be larger than the "balance" number reported by streams. In the case of one particular 2.8 GHz Pentium-4 system used for this example, the balance number is 16.5. The second code example above can't be extended directly, since that would require many more accumulator registers. Instead, we *block* the loop over i. (Technically, this is actually the second time we've blocked i, as the first time was the factor of 2.)

```
for(ii=0;ii<N;ii+=ib)
{
for(j=0;j<N;j+=2)
{
for(i=ii;i<ii+ib;i+=2)
{
acc00=acc01=acc10=acc11=0;
for(k=0;k<N;k++)
{
 acc00+=A[k][j+0]*A[i+0][k];
       acc01+=A[k][j+1]*A[i+0][k];
       acc10+=A[k][j+0]*A[i+1][k];
     acc11+=A[k][j+1]*A[i+1][k];
}
     B[i+0][j+0]=acc00;
     B[i+0][j+1]=acc01;
B[i+1][j+0]=acc10;
B[i+1][j+1]=acc11;
}
}
}
```

With this code, we can set ib to be anything we like, and the number of loads of the A matrix will be reduced by that factor. This freedom has a cost: we are now keeping a N * ib slice of the A matrix in the cache. So long as that fits, this code will not be limited by the memory system.

So what size matrix fits? Our example system, a 2.8 GHz Pentium 4, has a 16KB primary data cache. With ib=20, the slice of the A matrix in this code will be larger than the primary cache when N > 100. For problems larger than that, we'll need another trick.

That trick is reducing the size of the stripe of the A matrix by blocking the k loop, so that the stripe is of size ib x kb. Blocking the k loop means that the B array will be loaded and stored N/kb times, for a total of 2*N^3/kb memory transfers. A is still transferred N/ib times, for N^3/ib transfers. So long as

```
2*N/kb + N/ib < N/balance
```

the machine's memory system will keep up with the floating point unit and the code will run at maximum performance. The 16KB cache of the Pentium 4 is not quite big enough: we might choose ib=24 and kb=64, thus using 12KB of the cache -- we don't want to completely fill it, since the C and A arrays have to have some room to flow through. These numbers comes within 20% of the peak floating-point speed of the processor.

Here is the code with loop k blocked.

```
for(ii=0;ii<N;ii+=ib)
{
for(kk=0;kk<N;kk+=kb)
{
for(j=0;j<N;j+=2)
{
for(i=ii;i<ii+ib;i+=2)
{
if(kk==0)
     acc00=acc01=acc10=acc11=0;
else
{
          acc00=B[i+0][j+0];
          acc01=B[i+0][j+1];
    acc10=B[i+1][j+0];
          acc11=B[i+1][j+1];
}
for(k=kk;k<kk+kb;k++)
{
acc00+=A[k][j+0]*A[i+0][k];
        acc01+=A[k][j+1]*A[i+0][k];
acc10+=A[k][j+0]*A[i+1][k];
acc11+=A[k][j+1]*A[i+1][k];
}
    B[i+0][j+0]=acc00;
B[i+0][j+1]=acc01;
    B[i+1][j+0]=acc10;
    B[i+1][j+1]=acc11;
}
  }
}
}
```

The above code examples do not show the details of dealing with values of N which are not multiples of the blocking factors. Compilers which do loop nest optimization emit code to clean up the edges of the computation.

For example, most LNO compilers would probably split the kk == 0 iteration off from the rest of the kk iterations, in order to remove the if statement from the i loop. This is one of the values of such a compiler: while it is straightforward to code the simple cases of this optimization, keeping all the details correct as the code is replicated and transformed is an error-prone process.

The above loop will only achieve 80% of peak flops on the example system when blocked for the 16KB L1 cache size. It will do worse on systems with even more unbalanced memory systems. Fortunately, the Pentium 4 has 256KB (or more, depending on the model) high-bandwidth level-2 cache as well as the level-1 cache. We are presented with a choice:

- We can adjust the block sizes for the level-2 cache. This will stress the processor's ability to keep many instructions in flight simultaneously, and there is a good chance it will be unable to achieve full bandwidth from the level-2 cache.
- We can block the loops again, again for the level-2 cache sizes. With a total of three levels of blocking (for the register file, for the L1 cache, and for the L2 cache), the code will minimize the required bandwidth at each level of the memory hierarchy. Unfortunately, the extra levels of blocking will incur still more loop overhead, which for some problem sizes on some hardware may be more time consuming than any shortcomings in the hardware's ability to stream data from the L2 cache.

Rather than specifically tune for one particular cache size, as in the first example, a cache-oblivious algorithm is designed to take advantage of any available cache, no matter what its size is. This automatically takes advantage of two or more levels of memory hierarchy, if available[11].

5.2 Use of openmp

The OpenMP library is an API (Application Programming Interface) for writing shared memory parallel applications in programming languages such as C, C++ and FORTRAN. This API consists of compiler directives, runtime routines, and Environmental variables.

OpenMP divides tasks into threads; a thread is the smallest unit of a processing that can be scheduled by an operating system. The master thread assigns tasks unto worker threads. After wards, they execute the task in parallel using the multiple cores of a processor. Hence speed of execution can be increased thereby minimizing the amount of time taken to run the loops. An example for matrix multiplication using openmp is shown here[13].

```
printf("Thread %d starting matrix multiply\n",tid);

#pragmaompforschedule (static, chunk)

for(i=0;i<a_r; i++)
{
// printf("Thread=%d did row=%d\n",tid,i);
```

```
for(j=0;j<a_c; j++)
{
        for(k=0;k<b_c; k++)
        {
        c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
}
}
```

The time taken for executing this code segment for 500*500 matrix without using openMP is 1.59seconds and with using openMP is 0.55 seconds.

5.3 Storing the variables in registers instead of using memory

Optimization can be achieved by using vector SSE,SSE2,SSE3 intrinsics etc. SSE(Streaming SIMD Extensions) instructions operate in either all(SSE-packed) or the least significant pairs(SSE-scalar) of packed data operands in parallel. With SSE parallelism can be achieved by carry outing read-latency,execution and write-latency in one cycle by using prefetch, SIMD and streaming stores.

The execution is faster if the used data stored in registers by reducing access time.SSE provides eight 128-bit general-purpose registers each of which can be directly addressed using the register names XMM0 to XMM7. Depending upon the data length registers are used . If the data length is of 32bit and less each register can compute four instruction at one cycle (if used inside loop four instruction in one iteration), thus four times faster. Similarly data length is 16bit and less, eighth data can be stored in one register after execution result is eight times faster[12].

What is improved in the optimization?

1) Space optimizations - Reduces the size of the executable/object.

    1) Constant pooling

    2) Dead-code elimination.

2) Speed optimizations - Most optimizations belong to this category.

6. Conclusion

In this paper we have shown how to identify the areas in program (program segments) needed for optimization through profiling by using SAGE. And we have presented several optimization techniques namely loop nest optimization, SSE intrinsics and openMP for matrix multiplication. Thus, we can reduce time constraint which is a major issue in program analysis and development.

Acknowledgements

We thank our department head Mrs. Shantala C P for her encouragement and support. We also thank our family members and friends who endorsed to do the work.

References

[1] R. W. Butler (2001-08-06). "What is Formal Methods?". Retrieved 2006-11-16.

[2] C. Michael Holloway. *Why Engineers Should Consider Formal Methods*. 16th Digital Avionics Systems Conference (27–30 October 1997). Retrieved 2006-11-16.

[3] *link.springer.com/chapter/10.1007%2F3-540-45251-6_1*

[4] D. Richard Kuhn, Ramaswamy Chandramouli, Ricky W. Butler, "Cost Effective Use of Formal Methods in Verification and Validation".

[5] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, P. v.d. Stappen, "Model Checking for Managers", 6th International SPIN Workshop on Practical Aspects of Model Checking, Toulouse, France, 21 and 24 September 1999.

[6] P. E. Ammann, Paul E. Black, and William Majurski, "Using Model Checking to Generate Tests from Specifications", Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia (December 1998), edited by John Staples, Michael G. Hinchey, and Shaoying Liu, IEEE Computer Society, pages 46-54.

[7] J.R. Burch, E.M. Clarke, D.L. Dill, B. Misra, "Automatic Verification of Temporal Circuits Using Temporal Logic", IEEE Transactions on Computers, C-35, V. 12, pp. 1035 – 1044.

[8] Sage for Power Users: William Stein, February 13, 2012.

[9] Ella Bounimova, Patrice Godefroid, David Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production".

[10] SAGE for Newbies by Ted Kosan. http://creativecommons.org/licenses/by-sa/3.0/

[11] http://en.wikipedia.org/wiki/Loop_nest_optimization, Jan. 25, 2014 [Mar. 28, 2014]

[12] http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions, Apr. 1, 2014 [Mar. 28, 2014]

[13] Rifat Chowdhury "Parallel Computing with OpenMP to solve matrix Multiplication": http://biogrid.engr.uconn.edu/REU/Reports_10/Final_Reports/Rifat.pdf, Summer 2010 [Mar. 28, 2014]

[14]. www.afm.sbu.ac.uk

[15]. S. Merz, "Model Checking, a Tutorial Overview", F. Cassez et al. (eds): Modeling and Verification of Parallel Processes. Springer LNCS 2067, pp. 3-38. (2001)