Commission of the European Communities

****

ESPRIT III

PROJECT NB 6756

****

CAMAS

COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****

CAMAS-TR-2.1.1.6

SAD/Parasol II, Joined Report

****

Date: March 1995 — Review 5.0

ACE - Univ. of Amsterdam - ESI SA - ESI GmbH - FEGS - PARSYTEC -
Univ. of Southampton

Authors: Berry A.W. van Halderen
Jan de Ronde
P.M.A. Sloot

March, 1995
University of Amsterdam,
Faculty of Mathematics and Computer Science
Parallel Scientific Computing and Simulation group
Netherlands

1 Introduction

This report discusses the work done in the F2SAD and Parasol II subtasks for the elapsed review period. The F2SAD tool and the Parasol II tool are now being developed as an integrated toolset, with the possibility to use parts of it as a separate programs. The reason for this is that both tools will work on the same datastructures and need to share a lot of the same program code. The Parasol II program will perform simulations according to the same intermediate representation as F2SAD uses.

2 F2SAD

The release date for the F2SAD tool was set on end January 1995. Unfortunately, due to work on Parasol II which is vital for using F2SAD, this release date was not met. The tool is now ready for a release and will come with an extensive manual. The F2SAD tool has been tested to run properly on SunOS 4.1.x, Sun Solaris 2.2, IBM AIX, Silicon Graphics IRIX 5.2 and earlier versions also on HP-UX.

F2SAD has been subjected to a number of improvements:

The module in which the identifiers of the Fortran programs are stored has been fitted with a proper red&black balanced tree in stead of a normal lookup tree. This drastically improves performance; to give an indication, it previously took **15** minutes to process the PAM-Crash code, which now takes less than **2** minutes of CPU time.

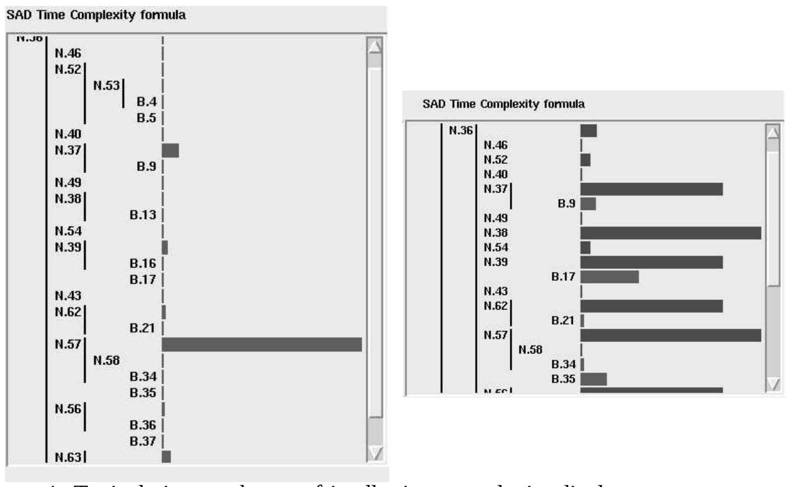
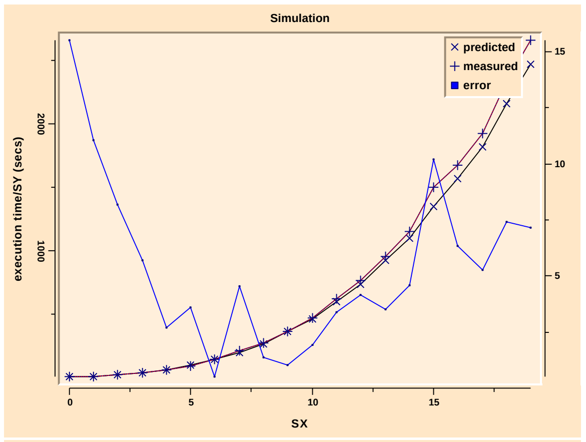


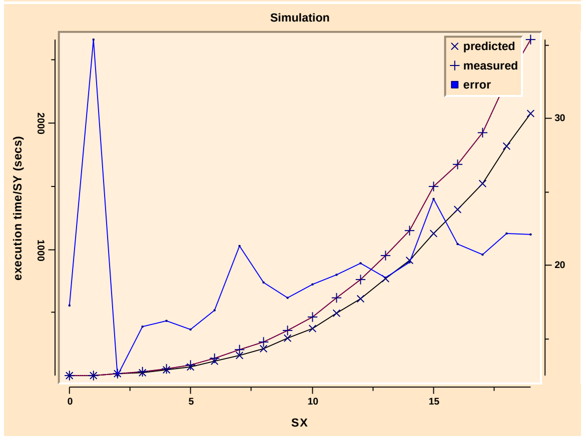
Figure 1: Typical view on the user friendly time complexity display.

The SAD time complexity formula can now be viewed in a user friendly way. This dramatically improves the usability of the formula, since we can now actually see which parts of a program contribute to the execution time.

A plot interface has been added using which data generated by the toolset can be viewed.

Momentarily we are experimenting with the modelling of caches. The model only incorporates the instruction cache. It is found that this parameter is quite sensitive. Look at the following two pictures for the MD1 Genesis benchmark (generated using the improved plot interface):





The upper figure shows the prediction and the error (as a percentage of the expected value) for an instruction cache size of 95 instructions and the lower for an instruction cache size of 100 instructions. With 95 instructions we get an error of approximately 8 percent while with 100 instructions per instruction cache we get an error of approximately 22 percent. Although these are both results which are very much predictable, we have the defect that the number of instructions that might fit in the cache in not such a rigid number. It quite depends on the quality of benchmarking and on the types of instructions executed (not all instructions occupy the same amount of bytes).

3 Parasol II

Time Complexity parameters

The SAD time complexity formula contains two abstract parameter classes. Machine constants which denote the computation time needed for operations and parameters which express the conditionals and loops in the program. To get any meaningful information out of the SAD formula both sets of parameters need to be actualized. The basic idea is then that we can experiment with the parameters to see the response in the evaluated, actualized formula. The machine parameters are actualized by the machine database, Parasol I. For the actualization of the algorithmic parameters there was still no conclusive method.

One way would be to define each parameter by hand, but for 3000+ parameters (in the program called "factors") for an application like PAMcrash this is unthinkable. So there must be another way of setting the factors. One way is to use data collected using test runs of applications. Using some specific sets of input data we run the application and generate output describing the flow of the program. Most compilers support this facility by generating a line-profiling executable, sometimes also called test coverage or simple profiling (look for the `-a` or `-xa` flag), which is normally used for this purpose.

Unfortunately line profiles (which count how many times a basic block is executed) are not very precise. We need to isolate the number of times a condition of a loop or if-statement evaluates to true, but there are numerous examples in which line profiles cannot give a decisive solution to this problem. We have experimented with a number of different experimental rules for extracting as much information from the lineprofile files as possible. This in order to actualize as much parameters in the time complexity formula as possible. We have settled on one method, which does not give a correct solution for all parameters. It will sometimes indicate that a loop is executed only once (or never at all) while the reality is very much different. This is the case for the PAM-Crash core code.

The parameters for the PAM-Crash code can now be actualized except for a single parameter, which is unfortunately the main loop of the program (a do-forever loop). This parameter cannot be automatically be determined, but that's quite exceptional. Almost all other parameters (over 3000) were also checked by hand on inconsistencies but none were found, which gives us enough confidence to continue with the current method.
But, the method used is, and will stay imperfect. It is possible that some parameters remain unset. The only way a conclusive method can be developed is by annotating the source code in the same way as a compiler would for line profiles. Needless to say that this is beyond the scope of this project. We have however now a proper working version for the PAM-Crash code.

We have been looking at other automated profilers like the WARTS toolset or the renewed profiling method of the GCC compiler. Unfortunately, tools within WARTS are too slow for larger programs and their operation is laborious and not very stable. The GCC/GPROF compiler and profiler have announced better profiling methods which would give a more detailed (probably up to the detail we would like) analysis of a program. But for these tools no formal specification of the end-result has been specified and the release date will be post CAMAS.

SAD level 3 simulator

We have developed an event simulator which simulates a network of processing nodes. The operation of this simulator is relatively simple and therefore very fast. On the one hand we have a pool of program tasks which "fire" instructions to the simulator. Such an instruction can be that the tasks would need to compute for a number of time steps. The simulator would then revoke the program task from the list of active tasks and places it back after that time.

A process can also generate synchronous receives or sends, in which case the task is again revoked, and placed back as soon as the receive or send has been completed. In addition to this a send instruction will also imply a load on the computer network. Such a load on the network is an abstract description for a claim of a piece of the bandwidth on certain communication lines in the computer network. Which lines in the network are involved and what the impact of the claim on the bandwidth is on the network as a whole is not part of the event simulator, but rather of the machine database. The load on the network is revoked as soon as the send or receive finishes.

The same rules apply for asynchronous receives or sends, except for the fact that the tasks are not revoked from the list of active tasks, so they are not prevented from generating more messages.

To determine when a receive or send has been completed the simulator will query the machine database with the abstract loads that are at that time imposed on the network. The machine database can then make an estimation when a send or receive instruction will terminate.

This simulator has been implemented and now needs to be integrated with the F2SAD intermediate representation.

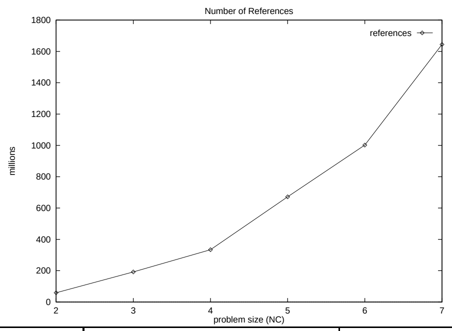
4 Other activities

Data cache influence

We have looked at the influence of the data cache on our familiar MD1 application, this was done using the WARTS[1] toolset.

The data cache simulation was done using the same target Sun Classic machine as in previous benchmarks.

```
Memory            : 32 MB
Model             : SPARCstation LX
At frequency      : 50 MHz
CPU               : Texas Instruments TMS390S10 (MicroSparc)
Data cache        : 2 Kb blocksize=16 1-way associative
Intsruction cache : 4 Kb blocksize=32 1-way associative
```



problem size	*number of references*	*cache miss rate*
2	59,240,677	0.0569
3	192,156,353	0.0561
4	333,945,838	0.0589
5	671,609,406	0.0571
6	1,001,928,083	0.0560
7	1,644,361,360	0.0562

These statistics show the data cache behaviour of the MD1 application. The average cache miss rate is 5 percent, which is quite low.
The tests cannot be run on a larger problem size than 7 due to limitations in the data cache simulator.

[1] This study made use of the Wisconsin Architecture Research ToolSet developed at the University of Wisconsin-Madison

Other applications

We have looked at other applications which are applicable for a study using F2SAD/Parasol II. Naturally, PAM-Crash from ESI and FAM from FEGS are the still in the running to be the final test case, but since these applications are rather large we should first look at other, smaller, but still real life programs.

Until now, only MD1 from the Genesis benchmarks has been used. We are expecting problems in the near future with this application since the parallel version is unstable. A second input program, which is based on a simulation method in which we have interest is required and for which a number of parallel solutions exist.

Relaxation algorithms are typically suited for investigation by means of the toolset. Implementations are of medium size, they are computationally intensive and they can be parallelized in multiple ways. Moreover, relaxation is used in many realistic applications and we will try to use one such a relevant application as input. The *shape from shading* problem can be used to extract height and slope from satellite pictures. We will try to use this application also in the near future and we are busy looking into the code and the parallel multi-grid method.