Dynamic instrumentation of loop blocks in parallel programs performance analysis

Aleardo Manacero Jr.
(aleardo@ibilce.unesp.br),
Davidson R. Boccardo,
José Nelson Falavinha Jr.,
Lucas dos Santos Casagrande, and
Henrique Jo Nakashima

[1] São Paulo State University - UNESP
Computer Science and Statistics Dept.
[2]
São Paulo State University - UNESP
Dept of Electrical Engineering
[3]
São Paulo State University - UNESP
Dept of Electrical Engineering
[4]
University of São Paulo
Computer Science Dept.
[5]
PCA Systems

Abstract. The computational solution for many applications demands a large amount of computing power, which is available through parallel computing. In the development of such systems their performance becomes a critical issue, since any loss of speed implies in the rise of costs with maintenance of IT personnel and machines. Hence, the application of performance measurement and analysis tools for these systems is quite important. This paper presents an alternative method for loop instrumentation based on Paradyn's dynamic instrumentation methodology, improving its granularity. A more focused measurement, at loop level, enables a greater accuracy on the results provided by Paradyn. A detailed description of dynamic loop instrumentation is provided, together with a few implementation aspects. Results achieved with loop instrumentation and some conclusions are also provided.

1 Introduction

In any system, using computers or not, performance is an important issue during its development; despite that, most system's developers do not put serious efforts into performance analysis and system optimization. For computer systems, a better performance means lower costs with the support of IT professionals and machines or less time required to achieve results. Therefore, the search for the best performance is crucial in systems that have either a heavy use or demand a high level of computing power. Parallel and distributed systems, usually classified as high performance systems, belong to this class.

The search for optimal performances demands methods and tools to measure and analyze performance data. Even in sequential systems these tasks impose several constraints such as what metrics to use, what parameters to measure and how much the tool execution interferes with the measured data. More constraints are needed when the measured system provides parallelism among processes, since parallel processes have to interact with each other.

Parallel programs require two other metrics for evaluation: speedup and scalability. The speedup provides information about how useful is the parallelization, while scalability tells how large a problem or system can be. Although these metrics are important, Sahni [12] indicates that the most relevant metric should be, under any condition, the execution time. This assumption is the main justification for the fact that almost all tools provide execution time (or other time related measures) as their main result.

One of the time related measures is the profile of the percentage of execution time spent on every function in a program, which is done by tools such as prof and gprof [11]. This function profile provides information about what functions should be optimized in order to achieve the best performance, becoming an important data for analysis. Indeed, the identification of bottleneck functions provide data and directions for the analyst to work on useful portions of the code.

A problem with most of the analysis tools is that they need iterative work during the measurement refinement. Since they work with post-mortem data, they have to go back to the measurement phase in order to perform a more focused analysis. Paradyn [7] provides a reasonable solution for this problem through its dynamic instrumentation. With Paradyn it is possible for the analyst to choose different metrics while the measurement is being processed, allowing for an easy modification of the code instrumentation if needed.

Since dynamic instrumentation is an interesting approach for performance measurement, it is useful to improve its accuracy. In its original configuration Paradyn collects data about the execution time of individual functions. These measurements are gathered in order to determine which functions are bottlenecks during the program's execution. This work improves Paradyn's accuracy through the instrumentation of loops inside bottleneck functions.

The next section describes some of the recent work on program instrumentation, being followed by a more detailed discussion of Paradyn. Then a general view of the loop instrumentation is presented. Results achieved with this instrumentation and conclusions compose the remaining sections.

2 Related Work

Performance analysis by software tools is a well-known strategy to describe characteristics of programs. Several tools perform analysis based on different aspects, such as the granularity of measured data, in order to find bottlenecks and derive appropriate optimizations. The basic mechanism to measure performance is benchmarking, where the program's code is instrumented and run in the actual system, in order to collect execution times. Other approaches, such as simulation or analytical modeling, are also used but provide less accurate results in general.

Some of more relevant benchmarking tools are SvPablo [2], and Kojak [9]. All of them allow user to gather a large variety of performance metrics and often provide graphical user interfaces to present results. However, most of them rely on static source code instrumentation and/or post-mortem performance analysis.

Some tools perform benchmarking with dynamic code instrumentation through frameworks like Dyninst [1], an API that dynamically inserts arbitrary code snippets into running applications. Tools like OMIS [14], DPCL [4], TAU [13], ToolGear [3], and Paradyn use this

mechanism for performance analysis. Other tools, such as DynTG [5], integrate dynamic instrumentation with a source browser and provides users with a fully interactive performance analysis environment.

All of these tools do not provide loop instrumentation in their proposals. Loop instrumentation appears on simulators that use the dynamic instrumentation approach such as IDTrace [16], which simulates systems based on the i486 family, and Metasim [8], which is a semi-cycle accurate simulator that works by gathering statistics on expected cache hit rates of routines and loops in an application.

As this brief review indicates, the tools that dynamically instrument loops are based on simulation. The reason for this is that dynamic instrumentation of loop blocks is hard to implement and implies several technical hazards, although providing better results. The following sections provide a description of Paradyn's operation and how the loop instrumentation could be added to it.

2.1 Paradyn Overview

Paradyn is a tool used for performance analysis of parallel and distributed programs. Paradyn instruments the binary code dynamically, in order to extract event traces without the need of user's modification on the program's source code or need of a special compiler. The instrumentation and performance analysis are made in real time, offering data for user analysis as they are obtained.

The code instrumentation is made through the creation of trampolines, where function calls are replaced by unconditional branches pointed to Paradyn instrumentations. Since these replacements occur without modifications in the remaining code, it is possible to maintain the program under execution at the same time that the replacements are made, enabling the dynamic instrumentation. In its current version, Paradyn instruments parallel programs, and therefore identifies its execution bottlenecks, at subroutine (functions) level.

The tool is composed of several modules, shown in figure 1:

- **Data Manager**, which manages the requisitions for data collection, receives and sends this data and, maintains and distributes information among the other modules;
- **Visualization Guide**, that performs the management of the visual interfaces and does the communication with the Data Manager;
- **User Interface**, which interacts with the user and controls the opened windows;
- **Performance Consultant**, which searches for bottlenecks and performs the communication between Data Manager and User Interface;
- **Paradyn Daemon**, which is the tool's back-end, being in charge of the initialization and control of the application processes. It is also in charge of their instrumentation and measurement.

3 Dynamic loop instrumentation

The dynamic loop instrumentation is a refined method for instrumenting code at execution time. Its goal is to offer a more accurate analysis of program bottlenecks. This instrumentation is based on Paradyn, taking advantage of the whole trampoline constructions that are

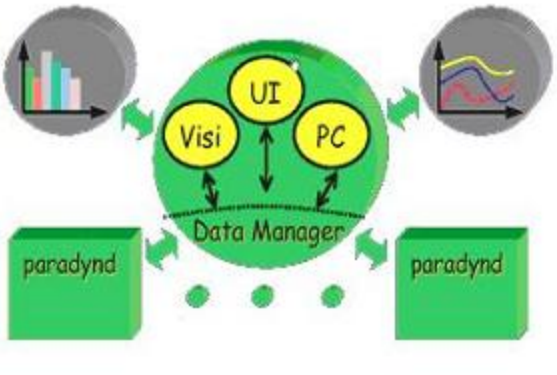

Fig. 1. Paradyn Standard Structure

already present in that tool. Although Paradyn provides a reasonable framework for loop instrumentation, there are several implementation problems that have to be solved in order to enable loop instrumentation. The following paragraphs describe such problems and their solutions.

Measurement's granularity

The choice of loops as a measurement grain derives from two aspects: the binary code organization and the performance issues of structural blocks. The former implies that only certain blocks can be instrumented. Formally, automatic instrumentation can be performed only at points where it is possible to determine the exact instructions that are the block's entry or exit points, restricting the choice to decision branches and loops. The latter one excludes decision branches, since they usually represent small amounts of sequential processing. Loops, on the other hand, are typically time consuming blocks and strong candidates for being bottlenecks. Therefore, the granularity for block instrumentation is restricted to the loop level.

Loop identification

As defined in the previous paragraph, a major problem in the instrumentation of inner blocks, such as loops, is the exact determination of entry and exit points. In any assembly language a loop can be characterized by one branch instruction pointed to some previous address, when compared to the address of the branch instruction. Therefore, the approach to find loops inside a function is to analyze the whole function's code and gather all branch instructions addressed to lower addresses. It is important to say that few situations, created by code obfuscation, would cause the identification of false loops. These situations are not relevant since their measurement will never indicate a bottleneck, representing only a small waste of instrumentation time.

Although the loop identification may be thought of as a simple process, there are some subtle differences among loops that can be detected. The loops that can be differentiated are:

– Single Loops - The loop's body does not contain other loops and is constrained by the addresses of the branch instruction and its target.

– Nested Loops - This loop contains inner (nested) loops. It is constrained by the addresses of the branch instruction and its target but have other pairs of such addresses inside its body (inner loops limits). Inner loops are defined as sons of the outer loop, while pairs of inner loops that do not have intersections are called brothers.
– Interleaved Loops - This appears when two loops share some portion of code but their branch and target addresses are interleaved, not nested. Although these loops may not be sons of an outer loop, they are also called brothers.

The identification and instrumentation of these loops imposes distinct actions from the system. A **single loop** is easier to identify and to manage since it is a monolithic piece of code. **Nested loops** are also easy to identify, although they demand the management of a parenthood list, in order to clearly distinguish all relationships between inner and outer loops. **Interleaved loops**, on the other hand, create several problems on their management, since their entry and exit points are not structured. Interleaved loops are very rare, but when found they are managed as a single unstructured loop by the instrumentation.

Hardware dependence

To accomplish the loop identification it is necessary to have access to the program instructions. Paradyn enables this access through a binary image of the code and its symbol table. Paradyn modules can read these images, decode the assembly instructions (using processor specific libraries), and perform the given instrumentation.

Fig. 2. Branch instruction format for the Sparc processor family.

The identification of branch instructions is performed through pattern matching for branch-like instructions, such as those for the Sparc architecture [10,15] given in figure 2. If the instruction is identified as a branch, the system determines its target address. If the target is a previous address, inside the function, there is a loop in this point.

Identification of instrumentation points

Every time that a loop is identified, it is inserted into a binary tree structure. The loop's *start* and *end* addresses, which are the keys for indexing, are stored in one node of that tree. These addresses, however, may not be the unique entry/exit points in the loop. In the binary code there could exist distinct branch instructions going outside the loop as well as entering the loop. All of these branches must be addressed as possible instrumentation points and, therefore, stored in the associated loop's node.

A distinct exit/entry point occurs at function calls inside a loop. One has to differentiate such points since the measured execution time for the loop should not include the time spent on the execution of the called functions. This approach follows the one used to measure the function's time, which also does not include the time spent on functions called from the

function body. Indeed, both approaches have to be the same in order to make the comparison between loop and function durations something meaningful. Therefore, it is mandatory to stop the measurement before calling a function and restart it upon its return. These points are also marked to be instrumented.

Displacement correction

The instrumentation points must be identified before actual instrumentation since all displacements in branch instructions must be recalculated, in order to accommodate the instructions inserted by the instrumentation. The new displacements are calculated by the composition among the number of instrumentation points that have to be inserted between the branch and its target and the size of each instrumentation point. This procedure is carried out for every branch inside the function that contains the instrumented loop.

A different problem is raised by some function calls in a few specific processors. If the call is made through a static address (set from the symbol table), the target remains the same and no modification has to be made. However, some processors, as the Sparc, provide the address of the called function by a displacement between the current address and the function address. In this case, this displacement must be recalculated since the address of the "call" instruction was moved by the instrumentations.

Instrumentation code

The actual loop instrumentation occurs through the insertion of calls to library functions that start or stop the time measurement. These functions use shared variables in order to provide the timing date for the Paradyn modules. The basic code of instrumentation, for Sparc processors, is shown in figure 3.

```
1. NOP
2. SAVE
3. SETHI      timerAddr>>10, %o0
4. OR         timerAddr & 0x000003ff, %o0, %o0
5. CALL       LoopStartTimer or LoopStopTimer
6. NOP
7. RESTORE
```

Fig. 3. Basic code of an instrumentation point.

All instrumentation must be carried out carefully, specially for function calls inside the loop and eventual modifications in the order of execution made to improve pipelines performance [6]. Each different situation implies distinct approaches to instrumentation codification. There are six types of instrumentation:

- STD_IN - inserted in the loop entries and after function calls, starting the measurement;
- STD_OUT - inserted in the loop exits and before function calls, stopping time measurement;
- FUNC_IN - inserted at the entry of a function, being similar to STD_IN, but starting the measurement of that function;

– FUNC_OUT - inserted at the end of function, also similar to STD_OUT, but finishing the timing of the function;
– OTHER_IN - inserted before branches which are additional loop entries, and adds code to prevent starting the instrumentation if the branch will not change PC value to the loop body;
– OTHER_OUT - inserted before branch instructions which are considered additional loop exits, adding code to prevent its execution in case the branch will not move to an address outside the loop.

Data collection
In order to get statistically valid timings it is necessary to collect several samples of loop execution times. The criteria used for data collection is to postpone the actual collection until the system reaches an observation limit and the execution is guaranteed to be outside the loop. This approach avoids one sample (the current one) becoming meaningless since it was interrupted before its actual finish.

Communication and integration with Paradyn
The integration with the original modules of Paradyn is performed through RPC calls. This method of communication is originally used in Paradyn and was slightly modified to accommodate loop information.

Since Paradyn analysis stops at function bottlenecks, the loop instrumentation is triggered only for those loops that are inside bottleneck functions, avoiding processing overheads while instrumenting lightweight functions.

Function relocation
As stated in the previous paragraphs, the instrumentation consists on inserting additional code into the function containing the loop under analysis. This task cannot be done inside the same address space used by the function. To circumvent this problem Paradyn uses trampolines for relocated functions, which have the effective instrumentation.

Therefore, the whole process of instrumentation occurs in a different address space, through the creation of an instrumented version of the function. This new function is reached, during program execution, by a branch targeted to its address. This branch instruction replaces the first instruction in the function's original address space, as seen in figure 4.

At the end of the time measurement, the original code of the function is restored and the relocated space is returned as free memory space. This enables future function relocations without expending too much memory.

4 Tests and results

The validation of loop instrumentation was performed through tests aimed to check the correctness of:

– Loop identification;
– Instrumentation point;
– Measurement accuracy.

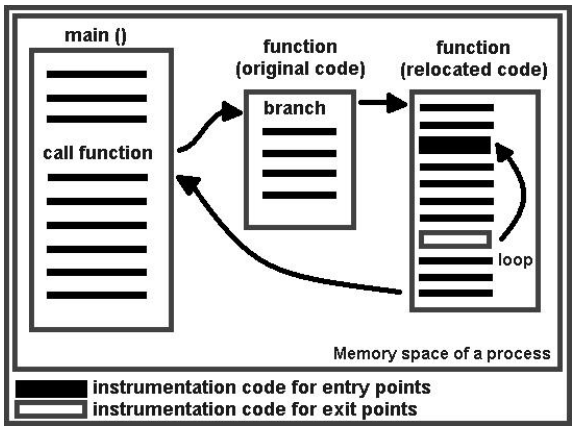

Fig. 4. Code Relocation

These tests are described in the following sections. For each aspect a small test was conducted, where the program under analysis could be easily controlled and verified. For sake the of simplicity, only these tests are presented here.

4.1 Loop identification

This verification consisted in building a sample code, containing several loops, and checking by hand if they were correctly identified by the program. One of such test codes is shown in figure 5, where there are several nested loops and a separated single loop. Although they do not appear in this example, interleaved loops were built with the use of *goto*'s and were also successfully tested.

```
Function
begin
    var K, J, W, I, T
    for I=0 until I<100
        J<-0
        do
            for W=0 until W<50
            for W=0 until W<50
                for T=0 until T<200
            J<- J + 1
        while J < 1
    K <- 0
    while K < 500000
        K <- K + 1
end
```

Fig. 5. Algorithm code for loop identification

The conducted tests produced outputs such as shown in figure 6, for the source presented in figure 5. Each line of this output presents loop's initial and final addresses. The line indentation represents the nesting level among each loop.

The addresses of each loop were verified, by hand, with the logical addresses of the actual loops, as read from the disassembled code. In all tests the loops were accurately identified.

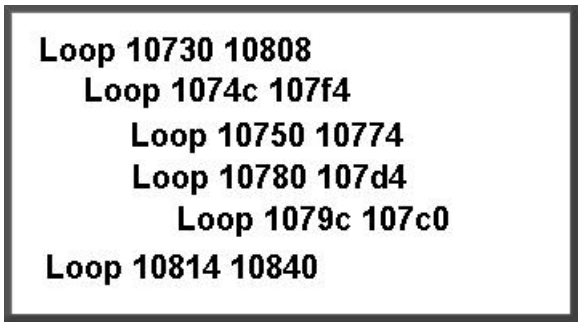

Fig. 6. Output of loop identification

4.2 Instrumentation points

In order to check for the correctness of the instrumentation points, the tests also used small programs and produced outputs containing the disassembled instructions of each instrumented loop. The representation of such outputs is very awkward, since they are quite large, even for small programs.

Verification was also made through manual inspection of the neighborhoods of call and branch instructions, as well as the recalculated displacements for their targets. The results were also very accurate.

4.3 Measurement accuracy

The previous validations (loop identification and instrumentation points) only assured that Paradyn was doing exactly what it was expected from it. The effectiveness of the loop measurements was not an issue when performing such tests.

To accomplish accuracy tests several benchmarks were conducted, measuring program execution times with the loop instrumentation done by the modified Paradyn. These measurements were compared to conventional benchmarking using Unix timing primitives.

A sample case appears in figure 7, that shows an algorithm of a part of the program used in the benchmarks presented here. In such simple algorithm there are three loops. Two of them are nested, while the third one appears later in a single structure.

The results achieved during the tests enabled the verification that the loop instrumentation is quite accurate, as shown in table 1. There, columns 2 and 3 show times measured with loop instrumentation, while column 5 shows times measured with Unix timing primitives. Column 4 shows the percentage of the function time that each loop consumes, which is the starting point to identify bottlenecks. From this table it is possible to notice that the

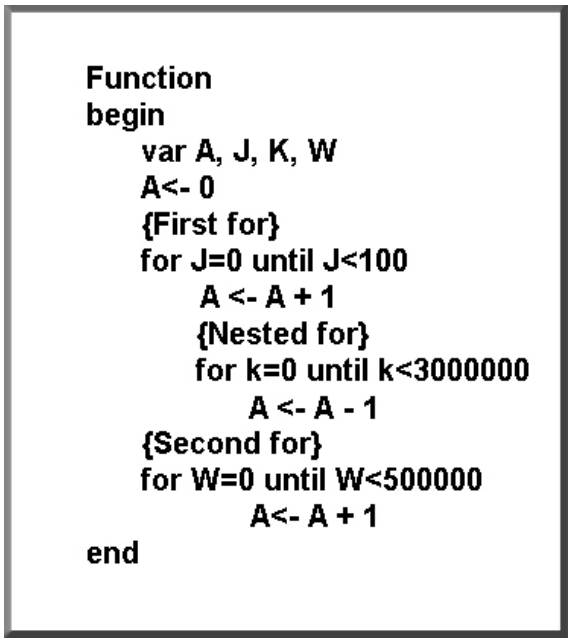

Fig. 7. Algorithm code for measurement times

error rate is lower for larger loops (less than 1%) than for smaller loops (up to 20% for very small loops). This is not a problem since the Paradyn's goal is the identification of execution bottlenecks, and small, fast, loops are not real candidates to be a bottleneck.

Table 1. Comparative times (seconds) between instrumented and Unix measurements.

Loop	Function duration	Loop duration	Percentage of function duration	Unix clock	Error (%)
1st for	9.683207	9.667192 s	99.83	9.720000 s	0.54
Nested	9.744907	9.727158 s	99.82	9.640000 s	0.90
2nd for	9.603596	0.016026 s	0.17	0.020000 s	19.9

The difference between the times measured by Paradyn for the nested loops is due, mostly, to instrumentation interferences and the distinct results in the measurement of function duration, which has to be performed again for every loop. These differences are not a problem, since the goal is the bottleneck identification. From this perspective, the modified Paradyn could correctly identify the "Nested" loop as the real bottleneck, since it could be accounted for much more than half of the "1st for" (actually it can be accounted for almost the whole duration of the "1st for").

5 Conclusions

The dynamic implemented loop instrumentation expands the functionalities of Paradyn's performance analysis. Although its original version achieves a good accuracy, reducing the granularity of measured blocks to the loop level is essential in the search for the real bottlenecks in the program. This would provide enough information to automatize some code optimization procedures, which could be later attached to Paradyn.

Such improvements in performance analysis tools are highly desirable since the amount of processing power and storage demanded by modern applications is rapidly rising. These new, high performance applications, are written by programmers that are not completely aware of the rules for efficient parallelization and probably are not experts on performance analysis. This makes it necessary that the analysis tools, such as Paradyn, become as automatic as they can be, including the choice for metrics or instrumentation levels.

Finally, the results achieved with loop instrumentation clearly show its efficiency. As previously stated the optimizations made over the identified bottlenecks should imply on sharper improvements of the program's performance.

Besides the interesting results already achieved, some new improvements may be added to Paradyn, such as:

- Implementation of techniques that enable automatic optimizations of the bottlenecks identified during the program analysis;
- Automation of the determination of the time spent inside the instrumentation, in order to have a more accurate value for code intrusion in each loop;
- Optimizations of the measurement and data collection functions;
- Modifications in the Paradyn's interface in order to accommodate the growth in the number of visual nodes by the loop instrumentation.

Acknowledgments

The authors must acknowledge FAPESP, that supported this research through grants (04/01340-0) and individual scholarships. They also have to thanks several people at Paradyn's group, specially Philip C. Roth, Eli Collins, and Matthew Legendre, who were always ready to answer configuration and implementation questions.

References

1. B. Buck and J. Hollingsworth. An api for runtime code patching. In *The International Journal of High Performance Computing Applications*, pages 14(4):317–329, 2000.
2. L. DeRose and D. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *In Proceedings of the International Conference on Parallel Processing (ICPP)*, Sept. 1999.
3. J. Gyllenhaal and J. May. Toolgear web page, http://www.llnl.gov/casc/tool_gear/. In *Lawrence Livermore National Laboratory*, Last accessed May, 2006.
4. T. Hoover L. DeRose and J. Hollingsworth. The dynamic probe class library | an infrastructure for developing instrumentation for performance tools. In *In Proceedings of the 15th International Parallel and Distributed Processing Symposium*, Apr. 2001.
5. J. Gyllenhaal M. Schulz, J. May. Dyntg: A tool for interactive, dynamic instrumentation. In *Lecture Notes in Computer Science, Volume 3515*, pages 140 – 148, Jan 2005.
6. M. M. Mano. *Computer System Architecture.* Englewood Cliffs, NJ: Prentice Hall,p. 310-319, 3nd ed edition, 1993.
7. B.P. Miller. *The Paradyn parallel performance measurement tool.* IEEE Computer, vol.28, n.11, p. 37-46, 1995.
8. PMaC Performance Modeling and Characterization. Metasim web page, http://www.sdsc.edu/pmac/metasim/metasim.html. In *San Diego Supercomputer Center*, Last accessed May, 2006.
9. B. Mohr and F. Wolf. Kojak - a tool set for automatic performance analysis of parallel programs. In *In Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 1301–1304, Aug. 2003.
10. R. P. Paul. *Sparc Architecture, Assembly Language Programming, & C.* Prentice Hall, 1994.

11. J.F. Reiser and J.P Skudlarek. Program profiling problems and a solution via machine language rewriting. In *ACM Sigplan Notices, vol. 29, n.1*, pages 37–45, 1994.
12. S. Sahni and V. Thanvantri. *Performance metrics: keeping the focus on runtime.* IEEE Parallel and Distributed Technology vol. 4, n.1, p. 43-56., 1996.
13. S. Shende and A. D. Malony. The tau parallel performance system. In *Submitted to International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.
14. V. Sunderam T. Ludwig, R. Wismuller and A. Bode. Omis | on-line monitoring interface specification (version 2.0). In *volume 9 of LRR-TUM Research Report Series. Shaker Verlag, Aachen, Germany*, 1997.
15. D. L. Weaver and Germond T. *The Sparc Architecture Manual.* Englewood Cliffs, NJ: Prentice Hall, version 9 edition, 1994.
16. J. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the aims toolkit. *Softw. Pract. Exper.*, 25(4):429–461, 1995.