Dynamic Program Instrumentation: Implementation Experience in HPUX

Ling Zheng

Technical Report #1348

May 1997

Dynamic Program Instrumentation: Implementation Experience in HPUX

Ling Zheng
Computer Sciences Department
University of Wisconsin, Madison

Abstract

Dynamic instrumentation is a new technique used in the performance tool Paradyn to provide efficient, scalable yet detailed data collection. It allows Paradyn to instrument a program to collect performance data on the fly. That is, the instrumentation can be inserted, removed, and changed at any time during program execution. This permits Paradyn to minimize instrumentation, instrumenting only what is necessary when it is necessary.

Paradyn's dynamic instrumentation environment consists of a code generator, a binary analyzer, and an instrumentation manager that allows code to be inserted and removed from the running program. This report describes some experience in implementing this environment in HPUX system.

1 INTRODUCTION

Dynamic instrumentation is a powerful technique that provides a scalable, efficient and detailed instrumentation. Unlike traditional binary rewriting tools, it allows instrumentation being inserted, changed and deleted at any time during program's execution. To exploit this technique, the Paradyn Performance Tool implements a dynamic instrumentation system that includes a code generator, a binary analyzer and an instrumentation manager. This report describes my experiences porting this system from Solaris operating system to HPUX operating system.

Dynamic instrumentation is conceptually simple, but complex in practice because of many architectural and system specific issues. The differences between machines and operating systems make the porting of this technique from one system to another system rather difficult. However, to make this technique widely available, it should work on most modern architectures, operating systems, providing the motivation to implementing this system on HPUX. At the same time, by describing the efforts putting into to port the technique from Solaris platform to HPUX platform, the portability of this system is demonstrated.

The report is structured as follows: In Section 2, I present an overview of dynamic instrumentation and its environment. Section 3 describes the major components of the system that are OS-dependent and required implementing effort. In Section 4, the detailed implementation of each component is given. Section 5 describes additional problems we encountered when doing the implementation. Finally, I give some performance evaluation of this system.

2 SYSTEM OVERVIEW

This section provides an overview of the dynamic instrumentation environment. First, the basic dynamic instrumentation operations are described, then the whole instrumentation system is presented.

2.1 The Basic Instrumentation Operations

To collect data, instrumentation code blocks are inserted into the program. A simple set of operations that can be used as building blocks to compute metrics is defined: *points*, *primitives*, and *predicates*. Points are interesting locations in

the application program where instrumentation can be inserted. Primitives are simple operations that change the value of a counter or a timer. Predicates are Boolean expressions that can be associated with primitives to control the execution of primitives.

By using these simple primitive operations, and combining them with predicates it is possible to create a large number of different performance metrics. For example, Figure 1 shows how instrumentation can be inserted into a single program to compute a metric that records the amount of time spent waiting for a message to be sent by the procedure `foo`. To compute this metric, four instrumentation code blocks are inserted. The first two are inserted into the procedure `foo` to keep track of whether `foo` is on the stack by *fooFlg* counter. The third and fourth code blocks are inserted into the message sending routine `SendMsg`. The value of *fooFlg* is then used as a predicate to control whether the two timer primitives will be called from `SendMsg` routine.

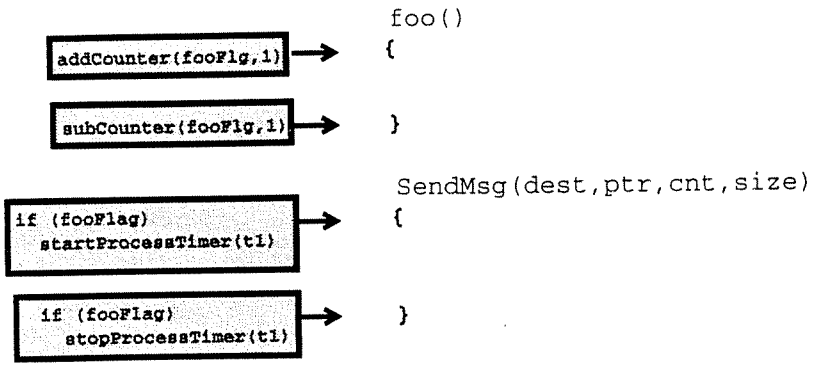

Figure 1: Sample Instrumentation Code

2.2 Instrumentation Design

The instrumentation is inserted into the program by tying the code generated from primitives and predicates together with the application. To do this, a set of small code fragments, called *trampolines*, is used. There are two types of trampolines: base and mini. Mini trampolines are used to hold the instrumentation code while base trampolines are used to connect the mini trampolines with application instrumentation points. There is one base trampoline per active instrumentation point, one or more mini trampolines per base trampoline. Mini trampolines are chained together much as done in Synthesis kernel [9].

The idea behind dynamic instrumentation is to let the instrumentation code to be executed along with program's execution. The instructions at the instrumentation point are replaced with branch instructions to jump to base trampoline. The original instructions at the point are relocated to the base trampoline. In addition, base trampoline contains instructions to save and restore registers, to call mini trampolines both before and after the relocated instructions, to jump back to application, and to keep track of instrumentation cost associated with this point.

2.3 Instrumentation System

An instrumentation environment implementing the dynamic instrumentation is show in Figure 3. The whole system consists of four main entities: the *parser*, the *code generator*, the *binary analyzer*, and the *instrumentation manager*. (This is a simplified version for the purpose of this report, but covers all the important details)

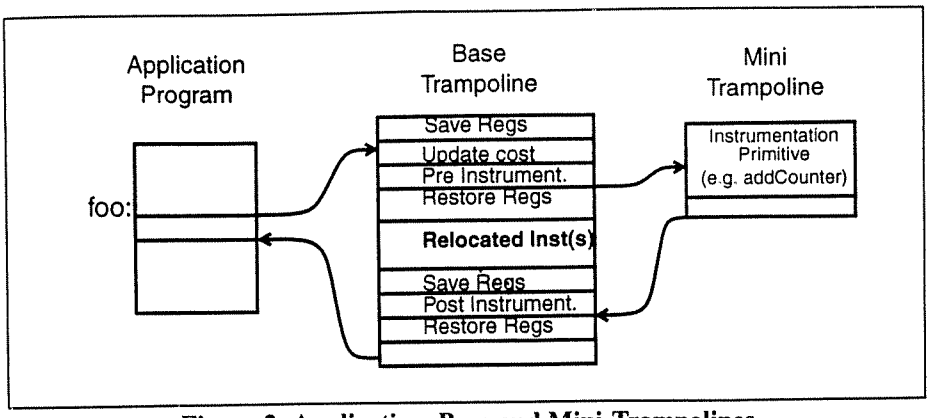

Figure 2: Application, Base and Mini-Trampolines

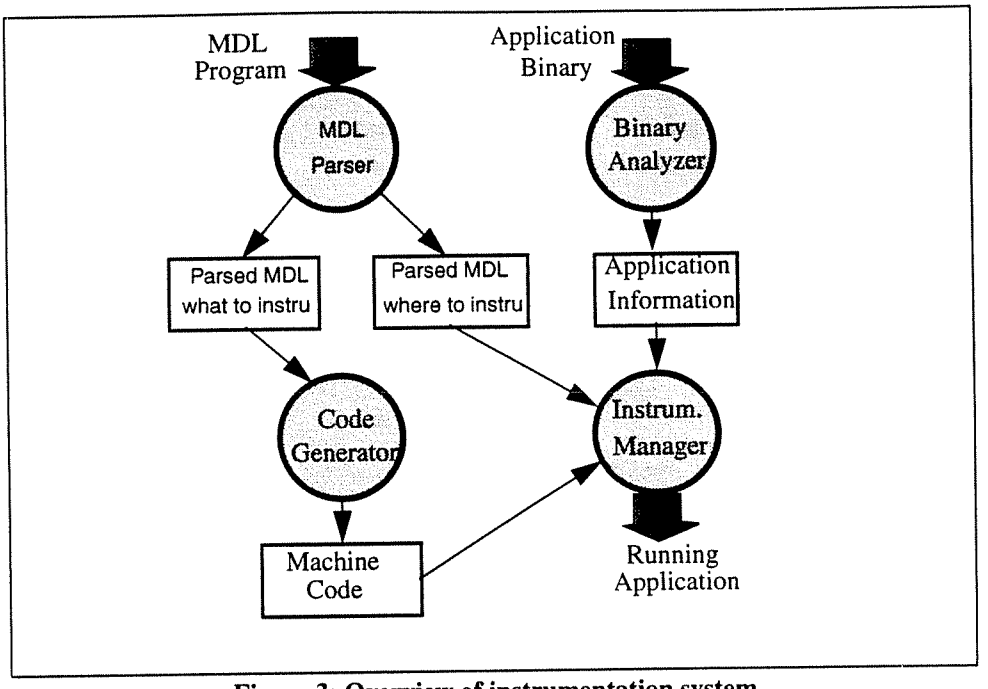

Figure 3: Overview of instrumentation system

Instrumentation requests are written in terms of performance metrics. The Metric Description Language (MDL) is a special language designed to describe all kinds of performance metrics easily. The *parser* is responsible for parsing the programs written in MDL. Two types of information are generated by the *parser*: what to instrument and where to instrument. The information of what to instrument is in the form of Abstract Syntax Tree (AST), an machine independent intermediate representation of the instrumentation code. The *code generator* is responsible to generate the machine code based on the AST. The *binary analyzer* performs a simple form of structural analysis to identify instrumentation points in the application. It takes the executable as input and extract all the necessary information for the instrumentation. Last, the *instrumentation manager* is responsible for inserting and deleting the instrumentation at program's running time. With the information provided by the other three entities, it completes the instrumentation processing by editing the application's image in memory.

3 IMPLEMENTATION REQUIREMENTS

The instrumentation system was first developed on Solaris platform. To port it to HPUX, the first task was to identify the parts of system that required major changes. The modules that needed to be changed are briefly listed bellow, and described in more detail in subsequent sections.

- Code Generator. Solaris and HPUX operating system are running on SPARC and PA-RISC architectures respectively. The instruction sets provided on these two machines differ widely both in formats and functionality.
- Binary Analyzer. The format of executable objects in these two systems are different as well. This includes information in sections containing text segment, symbol table and others.
- Instrumentation Manager. The instrumentation manager depends on the functionality provided by the operating system. For process control, Solaris has an efficient special purpose */proc* file system. HPUX has no such counterpart (it uses *ptrace*( )).

4 IMPLEMENTATION EXPERIENCE

4.1 Code Generator

The responsibility of code generator is to generate machine code from a machine independent intermediate representation of the instrumentation code, Abstract Syntax Tree (AST). While the AST is traversed, a code generation routine is called for each node type. The code generator here is different from most traditional code generators in that it directly generates the binary representation of machine code instead of assembly code. There are three key differences between SPARC instruction set and HPUX instruction set that affect the code generator.

- SPARC uses the concept of "register windows" to do register allocation. By moving the window up and down, different physical registers are accessed. PA-RISC uses 32 general integer registers and register allocation needed to be taken care of by the code generator
- SPARC provides instructions such as *save* and *restore* to perform many of tasks necessary to make to a procedure call. This is not the case for PA-RISC. Instead, the procedure call is accomplished via a software convention. This provides more flexibility but adds some complexity to the code generator as well.
- The formats of the PA-RISC instruction set are more complicated than SPARC instruction set. To generate an instruction, some rather subtle computations are needed to figure out the value for each field of this instruction due to encoding of the operand addresses.

The code generator only generates a subset of instruction set, including memory reference instructions, branch instructions and computation instructions. The code generated keeps track of performance data such as the number of times this point has been executed, the value of parameters passed to one function. Figure 4 and Figure 5 show two simple examples of the code generated by the Paradyn HPUX code generator.

```
// Instrumentation code ("add counter" primitive)
//    Load counter
minitramp:      ldil 4070e000,ret1
minitramp+4:    ldo 340(ret1),ret1
minitramp+8:    ldw 0(sr0,ret1),ret1
//    Increment counter
minitramp+12:   addi 1,ret1,ret1
//    Store counter
minitramp+16:   ldil 4070e000,r3
minitramp+20:   ldo 340(r3),r3
minitramp+24:   stw ret1,0(sr0,r3)
// Branch to either base trampoline or next mini-trampoline
minitramp+28:   b,n basetramp+52
minitramp+32:   nop
```

Figure 4: Mini trampoline for primitive addCounter

```
// Instrumentation code ("startProcessTimer" primitive)
//    Save parameter register
minitramp:      stw r26,-a4(sr0,sp)
//    Set up argument (the address of timer)
minitramp+12:   ldil 4070f800,ret1
minitramp+12:   ldo 360(ret1),ret1
minitramp+12:   stw ret1,-24(sr0,sp)
minitramp+28:   ldw -24(sr0,sp),r26
//    Call DYNINSTstartProcessTimer
minitramp+16:   ldil 5800. r31
minitramp+20:   ble,n 5b8(sr4,r31)
minitramp+24:   nop
// Restore parameter register
minitramp+28:   ldw -a4(sr0,sp),r26
// Branch to either base trampoline or next mini-trampoline
minitramp+28:   b,n basetramp+52
minitramp+32:   nop
```

Figure 5: Mini trampoline for primitive startProcesTimer

4.2 Binary Analyzer

The binary analyzer is responsible for identifying instrumentation points in the application program. It works directly on executable objects and independent of compiler and language systems used. An executable is processed in two steps. In the first step, it reads the infomation from the symbol table and obtains a list of symbols along with its name, type, starting address and size. In the second step, it scans the binary image to find out the instrumentation points in each function.

4.2.1 Symbols

The common object format used in HPUX system is called SOM. SOM is used as a common representation of code and data generated by any compiler that generates code for PA-RISC. It consists of a main header record, an exec auxiliary header record, symbol dictionary, subspace dictionary, and other optional components. The header record is stored at the beginning of the executable files. The SOM is processed in three steps:

1. The infomation of header record is read. First, the executable file used is verified to be of PA-RISC format by comparing of the value in the field system id and file type with pre-defined magic numbers. Then the infomation about other components such as location and length is retrieved.
2. From the auxiliary header record, find out the location and size of text segment and data segment.
3. Obtain the symbol records from symbol dictionary. There are sixteen symbol types, however, for this instrumentation system, only four type of symbols is used. These are DATA, CODE, ENTRY, PRI_PROG. The PRI_PROG symbol is used for the Fortran program to identify the entry point of an application (for C and C++, the entry point of an application is function "main", but this is not necessarily true for Fortran programs).

While SOM provides much useful infomation, it is not complete for our purposes. One type of infomation missing is the size of each function, which is needed when we scan the binary image. We obtain this infomation by sorting all the symbols by address and compute the size of each symbol by subtracting the address from the address of next symbol.

The other important piece of infomation not provided in this format is the list of modules (source files) in this object file. This infomation is obtained from the symbol debugging table. Unfortunately, the format of symbolic debug infomation is different for different compilers, e.g. Gnu CC and native CC. The format used by Gnu CC is the standard "stabs" debug format. Unfortunately, the format used by HP native CC is proprietary and confidential to HP. The debugging symbols generated by GNU CC are put in the subspaces GDB_STRING and GDB_SYMBOL; while the debugging symbols produced by HP native CC are put in the subspaces $GNTT$, $LNTT$, and VT. By looking up these names in the subspace dictionary of the SOM, the analyzer finds out which compiler compiled this executable file. If all these subspaces exist, it means part of this executable file is compiled by Gnu CC and part of it is compiler by HP native CC. In this situation, we need to process all the subspaces.

In addition to the above, the subspace $UNWIND_START$ is processed. This subspace contains an unwind table holding the stack unwind information. Each entry in the unwind table contains two addresses that are the starting and ending address of a function as well as other infomation about the frame and the register usage of that region.

4.2.2 Instructions

After we have a list of functions, we scan the instructions in the segment one by one to find the instrumentation points for each functions. The instrumentation points contain the instructions that are eligible to be relocated to other places. There are three type of instrumentation points: function entry, function exit and call sites in every functions. For Solaris platform, in most cases only one instruction needs to be identified. However, for HPUX platform, at least two consequential instructions is needed for each instrumentation points. This complicates the implementation of instrumentation system:

- The function entry point is defined as the first two instructions within this function. If the second instruction is

a branch instruction, the third instruction is also included for this instrumentation point.

- The call site point is identified by the Branch and Link (*BL,n*) instruction. The instrumentation point consists of this *BL,n* instruction and the instruction in its delayed slot. With the address of this branch instruction and offset infomation contained in the branch instruction, the target address of this branch is computed. This address is checked to make sure that only branch instructions branching out of this function will be counted as call site points. For this type of instrumentation points, if there's another branch instruction in the delayed slot of the branch instruction, we disable this instrumentation point to keep the analyzer simple.
- The function exit point is identified by the Branch Vectored (*BV,n*) instruction. The point associated with this instruction includes another instruction which is right before this instruction. The reason is that there's no guarantee that one more instruction slot would be available after this exit instruction. The instruction following the *BV,n* instruction is only available and included when no nullification is specified. If nullification is specified, the next instruction might not belong this function and is illegible to be relocated at this point.

After identifying all the instrumentation points, this function is scanned again to make sure that (1) there are no direct jumps in this function branching to the middle of these instruction sequences and (2) there is no overlap of instructions among these points.

4.3 Instrumentation Manager

The instrumentation manager is responsible for the insertion of instrumentation and deletion from the application process. This is done by tying the trampolines together with and separating the trampolines from the application. To insert instrumentation, the application is stopped at first, then the code and data is installed using operating system debug facilities. Solaris uses */proc* file system, HPUX uses *ptrace()* system call. After these operations, the instructions at the instrumentation point are replaced with branch instructions to jump to the base trampoline and the original instructions are relocated to the base trampoline. To delete the instrumentation, the mini trampolines corresponding to the instrumentation to be deleted are removed from the chain of trampolines.

4.3.1 Instrumentation Insertion

The insertion of instrumentation includes creating trampolines, relocating instructions and modifying the original instructions to jump to the base trampolines. There are a few issues need to considered for the jump instructions and instructions' relocation.

4.3.1.1 Inter-module branching

For HPUX system, the trampolines are put into the data segment. Thus, the execution of instrumentation code along with application process requires the branching from text segment to data segment and from data segment back to text segment. In Solaris, these two segments are contiguous to each other. However, in HPUX, the virtual address of them are far away to each other and address in one segment could not be reached from the other by one simple PC-relative branch instruction. Instead, the following instruction sequence is used to handle the jump:

```
foo:   ldil l%basetramp,r31
foo+4: ble,nbasetramp(sr5,r31); branch and link, with return in r31
```

```
// Create a new stack frame and save registers
// Link register(rp),frame register(r3),return register(ret0),r31 are saved
basetramp:      addi 80,sp,sp
basetramp+4:    stw r31,-7c(sr0,sp)
basetramp+8:    stw rp,-78(sr0,sp)
basetramp+12:   stw r3,-74(sr0,sp)
basetramp+16:   stw ret0,-70(sr0,sp)
// Update cost
basetramp+20:   ldil 4070d800,ret1
basetramp+24:   ldo 520(ret1),ret1
basetramp+28:   ldw 0(sr0,ret1),ret1
basetramp+32:   addi 13,ret1,ret0
basetramp+36:   ldil 4070d800,ret1
basetramp+40:   ldo 520(ret1),ret1
basetramp+44:   stw ret0,0(sr0,ret1)
// Branch to mini-trampoline (pre-instrumentation)
basetramp+48:   b,a   minitramp
// Restore registers
basetramp+52:   ldw -70(sr0,sp),ret0
basetramp+56:   ldw -74(sr0,sp),r31
basetramp+60:   ldw -78(sr0,sp),rp
basetramp+64:   ldw -7c(sr0,sp),r31
basetramp+68:   addi -80,sp,sp
// Space for relocated instruction(s)
basetramp+72:   stw rp,-14(sr0,sp)
basetramp+76:   copy r3,r1
basetramp+80:   nop
// Create a new stack frame and save registers
basetramp+84:   addi 80,sp,sp
basetramp+88:   stw r31,-7c(sr0,sp)
basetramp+92:   stw rp,-78(sr0,sp)
basetramp+96:   stw r3,-74(sr0,sp)
basetramp+100:  stw ret0,-70(sr0,sp)
// Branch to mini-trampoline (post-instrumentation)
basetramp+104:  nop
// Restore registers
basetramp+108:  ldw -70(sr0,sp),ret0
basetramp+112:  ldw -74(sr0,sp),r31
basetramp+116:  ldw -78(sr0,sp),rp
basetramp+120:  ldw -7c(sr0,sp),r31
basetramp+124:  addi -80,sp,sp
// Branch to the application code
basetramp+128:  addi -4,r31,r31
basetramp+132:  be,n 0(sr4,r31)
```

Figure 6: Base trampoline

To jump between segments correctly, the space register used needs to be specified correctly in the *ble, n* instruction (space register 5 for jump to the data segment and space register 4 for jump to the data segment). In the above instruction sequence, r31 is used for calculating the target address. This register is defined to hold the millicode[1] return pointer. It is assumed available for our purpose since we are not interested in instrumenting any millicode. In this example, after the *ble,n* instruction is executed, the value of foo+12 is saved in the r31. This value is later used in

1. Millicodes are special purpose routines that are tailored for performance. These routines are written in the PA-RISC assembly language and follow the millicode calling conventions[2].

the base trampolines. In the following two paragraphs, I describe the inter-segments jumps (jumps between application's text segment and data segment) used in this system.

In the base trampolines, there are two places where the instrumentation code finishes in the data segment and jumps back to application's text segment. For function entry and call site points, the last two instructions at the base trampolines jump back to the application's text segment as shown in Figure 6. The value stored in r31 is first adjusted to the correct target address, then the branch is taken. For the function exit points, the relocated instruction *BV,n* will branch back to the application. To make it work correctly, the *BV,n* instruction is changed to *BE,n* instruction with the space register specified as sr4.

In both base and mini trampolines, there are situations where we need the jump from the application's text segment back to the instrumentation code in the application's data segment. Here are two examples: (1) for the call site point, the call instructions are moved into data segment to make the procedure call; (2) we want the instrumentation code in the mini trampolines to call application's procedure. In both cases, when the procedure call finishes, the inter-segment jump is necessary to jump back to data segment. Two solutions are applied in different situations.

- The first solution is used for the call site points all the time and used for the procedure call in the mini trampoline some times. It could be applied to all the situations but is a bit more expensive. We put an assembly routine between the trampoline and application. The trampoline calls the assembly routine and passes the address of procedures. With that address, the assembly routine calls the procedure in the application's text segment, then the application returns to trampoline after the procedure call is finished.
- The second solution is to modify the instructions in the procedure called. For these procedures, appropriate instructions are tailored to be suitable for inter-segments calling conventions. Since these procedures are no longer following standard calling conventions, they are ineligible to be called by any other procedures in the application's text segment. In our implementation, this solution is only applied to the procedures in Paradyn's library (dynamically) linked to user's application (these procedures are not to be called by any other procedures except the instrumentation code in the data segment). The advantage of this solution is that it does not add any extra instrumentation overhead.

4.3.1.2 Instructions relocating

Most instructions do not require any changes to relocate them from one place to another. However, this is not true for PC-relative branch instructions. For these instructions, we need to calculate the target address and then modify the instruction so that the target would stay the same after the instruction is relocated. One problem on HPUX system is that after a branch is relocated from the text segment to the data segment, the target address would be too far to reach for simple branch instruction. The problem is solved, again, by taking advantage of the value in the r31. As mentioned before, r31 contains an address related to the instrumentation point. When the branch instruction is relocated to the base trampoline, instead of putting a branch instruction in the base trampoline, some other instructions is generated to change the value in r31 by the corresponding offset in the branch instruction. Then when the program executes the last two instruction to go back to the application, it jumps to the target of the branch instruction relocated and creates the same effects as the original branch instruction. For conditional branch instructions, it is a little bit more tricky. Depending on whether this branch will be taken or not, different adjustments will be made to r31.

The other issue relating to relocating multiple instructions is that the program might be executing in the middle of the instruction sequence at the moment the instrumentation is inserted. This requires a stack walking operation. If the program is simply resumed after the instrumentation, problem would arise since only part of the branch instruction sequence will be executed. The solution here is to delay the instrumentation insertion at this moment and insert a breakpoint at the end of the sequence. After the program is resumed, it will be stopped again at the breakpoint. The instrumentation is inserted since it is guaranteed to safe then.

4.3.2 Instrumentation Deletion

The deletion of instrumentation is implemented by modifying the jump between base trampoline and mini trampolines to bypass the mini trampoline associated with this instrumentation deletion request. At the same time, the memory block containing the mini trampoline is marked as available for future use.

The issue here is that when the instrumentation is deleted, it is necessary to make sure that this instrumentation code is not currently active. This is done by tracing the stack and comparing the PCs in all the frames with the region of this instrumentation code. The stack tracing is an platform dependent operation. The implementation on HPUX platform is described in the subsequent section.

5 OTHER IMPLEMENTATION ISSUES

While porting the dynamic instrumentation from Solaris platform to HPUX platform, many other issues need to be considered. For example, the interface of many system library routines are different on these two operating systems. Some routines on Solaris are not available on HPUX, such as *getrusage()*. We then need to find some other routines that provide similar functions of those, such as *pstat()*. In the following subsections, two operations needed for dynamic instrumentation are described. Both of them require some non trivial porting effort.

5.1 Stack tracing

The most important use of stack tracing arises when we are inserting and deleting the instrumentation. On Solaris, stack tracing works as follows: first, the value of program counter and stack pointer of the application is read; then with the stack pointer, we read the value of stack pointer and program counter for the previous frame; last, when the stack pointer for previous stack frame is zero, it means this is the last active stack frame and this processing finishes. However, on HPUX, this will not work because the value of next stack frame is not necessarily stored on the stack. Instead, the implementation on HPUX uses the infomation read from executable file's subspace $UNWIND_START$ to assist the stack tracing. All the records read from this subspace are put into a table and sorted by their starting addresses. This processing works in the following three steps on HPUX:

1. First, the value of program counter is read in the following way: read the save_state structure via a ptrace call to find out whether the application is currently executing in a system call. If true, read the PC from r31; otherwise read the value of PC from r33(the PCOQ_HEAD). At the same time, the value of stack pointer is read.
2. With the PC value, we find the corresponding entry in the unwind table using a binary search of starting address of all the entries. If no entry found or an entry is found but this entry is associated an interrupt frame, the stack tracing processing quits. Otherwise, find the frame size in the entry and compute the new stack pointer with the current stack pointer and the frame size. Also with the current stack pointer we obtain the value of program counter of the previous frame.

3. On HPUX platform, the value of stack pointer will not necessarily be zero. The stack tracing finishes by checking if correct frame is associated with procedure _start or $START$.

5.2 Inferior procedure call

The inferior procedure call means triggering an application procedure call from the Paradyn daemon. Here is one example of why it is necessary. To measure how much time spent in one particular function, we instrument this function by inserting the code to start a timer at the entry point and stop the timer at the exit point. If while we are inserting the instrumentation code, the program counter is currently within this routine. The instrumentation code to stop the timer at the exit point will be executed while the instrumentation code to start the timer at the entry point is unfortunately missed. A solution to this problem is to make the application execute the inserted code (in this example, the code to start the timer) before it resumes. This is accomplished by the inferior procedure call.

To do the inferior procedure call, first the application is stopped. Then the registers and necessary context are saved to the stack. After that, the code to be executed is generated and appended with a trap instruction at the end of the trampolines. After changing the program counter from the application to the first instruction of this code block, we resume the application. The application runs until it finishes the instrumentation code and hits the trap instruction. Now the instrumentation manager takes over, switches the PC back to the original value in the application, restores the context from the application and resumes the application again.

On HPUX, the implementation is a little bit trickier. There are two situations that need to be handled separately: when the application is running in a system call and when it is not. When the application is not currently in the system call, we use an assembly routine DYNINSTdyncall (the same one as millicode $$dyncall) to do the inferior procedure call. Instead of changing the PC directly from application to trampoline that contains the generated code, the PC is changed to this assembly routine and the value of r22 is set to the address of trampoline. In this way, the application could execute the instrumentation code correctly. The other trick is played when the application hits a trap instruction. Instead of changing PC then, the application jumps back to the application by execute two other instructions as the following. Before the execution of these two instructions, the appropriate value for space register and program counter are written into register 21, 22.

```
mtsp r21, sr0
ble,n 0(sr0, r22)
```

When the application is currently running in a system call, it is treated in a slightly different way. Most of the details are the same, except that the assembly routine is no longer needed and the PC could be directly set to the location of trampoline. The reason is that when the system call finishes, the kernel will set up the space register correctly based on the value in r31. Calling $$dyncall will not work because the value in r22 would be changed while system call finishes.

6 PERFORMANCE MEASUREMENTS

The performance cost for the instrumentation is the time it takes to go from the user code to the trampolines and back. The cost of the instrumentation code itself (the part that calculates the metrics) is specific to the semantics of the metric description. The more specifically the performance data you want to gather, the more expensive the instrumentation code would be.

	SPARC, 110MHz, microSPARC II	PA-RISC, 50MHz, HP 9000/715
Call to empty procedure	87ns	261ns
+ base trampoline w/jump (no saves/restores)	209ns (2.4)	633ns (2.4)
+ base trampoline w/jump (with saves/restores)	504ns (5.8)	1374 (5.3)
+ empty mini-trampoline	556ns (6.4)	1632ns (6.3)

The figure above shows some simple measurement of the cost. The relative instrumentation costs of the SPARC and HP are almost the same. Although the instruction set and procedure calling conventions are widely different for these two architecture, the instrumentation overhead of those instructions are surprisingly the same.

7 CONCLUSION

This report described the implementation experience of the dynamic instrumentation on HPUX platform. The system consists of three major modules which need substantial efforts: Code Generator, Binary Analyzer, Instrumentation Manager. The contribution of this report is in pointing out the problems in porting the dynamic instrumentation system. This may be of use to others who are undertaking similar tasks.

REFERENCES

[1] Hewlett Packard Co. PA-RISC 1.1 Architecture and Instruction Set Reference Manual. 1990.

[2] Hewlett Packard Co. PA-RISC Procedure Calling Conventions Reference Manual. 1991.

[3] J.K. Hollingsworth, R.B. Irvin, and B.P. Miller. The Integration of Application and System Based Metrics in a Parallel Program Performance Tool. *3rd ACM Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA (April 1991).

[4] J.K. Hollingsworth, B.P. Miller, and J.M. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. *Scalable High Performance Computing Conference*, Knoxville, TN (May 1994).

[5] J.K. Hollingsworth and B.P. Miller. Dynamic Instrumentation API. *Computer Sciences Technical Report*, `http://www.cs.wisc.edu/~paradyn/dyninstAPI.html` (September 1996).

[6] J.K. Hollingsworth, B.P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. Tech.l Report, Comp. Science Department, UW-Madison. 1996.

[7] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA (June 1995).

[8] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11), November 1995.

[9] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. 15th Symposium on Operating Systems Principles, Copper Mountain, CO (December 1995).

[10] Pure Software. United States Patent 5,193,180, March 1993.

[11] Sun Microelectronics. UltraSPARC User's Manual. 1996.