JiTI: a Robust Just in Time Instrumentation Technique

Michiel Ronsse and Koen De Bosschere
Department of Electronics and Information Systems
Ghent University, Belgium
{ronsse|kdb}@elis.rug.ac.be

Abstract

In this paper, we describe *JiTI*, a novel technique for instrumenting program binaries. The technique correctly deals with programs that contain traditionally hard to instrument features such as data in code, code in data, and self-modifying code. The technique does not require reverse engineering, program understanding tools or heuristics about the compiler or linker used. The basic idea is that a running process is cloned in memory, and that the cloned process is completely instrumented (code + data). By using the code of the instrumented process (clone) on the data of the original process, we can guarantee a correct instrumentation of the full code while keeping the process data untouched. *JiTI* has been completely implemented for SPARC processors and is used in a data race detector.

1 Introduction

Binary modification is the ultimate language-independent technique to change programs. A binary modification tool reads a binary program, analyses it, modifies it, and creates a new binary ready for execution. The modification can range from inserting profiling code, optimizing the code, translating it to a new architecture, etc. The method works on machine language instructions, and is therefore programming language independent. Since it does not depend on the original program sources, it can also be applied to libraries, programs consisting of parts written in different programming languages, etc. In order words, anything that can be understood by the underlying processor, can also be understood by the binary modification tool.

There are several successful applications of binary modification systems around: examples are the basic block counting tools QPT [Ball92] and EEL [Laru96] for Sun SPARC machines, ATOM [Sriv94] and Alto [DB96] for Digital Alpha machines, ETCH [Lee] for Intel machines running Windows NT and Paradyn for multiple architectures [Mill95, Zand99, Xu99].

Besides the nice features of binary modification, there are is also a dark side. Instrumenting binaries basically boils down to inserting instruction in the binary, requiring relocation of data and code. In order to do this, one must be able to disassemble the program into *basic blocks* and *control flow graphs*. This requires a sophisticated analysis of the binary, often requiring an enormous amount of resources (both in time and space). Moreover, in order to be able to modify a binary program, one should exactly know what the program is doing. Without extra information about

the binary (e.g., the compiler that generated it), it can be very hard to recognize data in code, code in data, constructs like user-level context switches, self-modifying code, hand-written assembly language, etc.

JiTI offers a solution that effectively deals with this kind of unconventional code. The tool has been implemented for SPARC processors (tested on MicroSPARC, HyperSPARC, SuperSPARC and UltraSPARC) and was successfully used as the underlying mechanism for RecPlay, a data race detector [Rons99]. In the data race detector, *JiTI* is used to instrument every memory operation in a program.

The next section contains a detailed description of the implementation of the *JiTI*-concept for the SPARC architecture. An evaluation of the implementation is given in section 3. The paper ends with an overview of related work and the conclusions.

2 *JiTI* for the SPARC

Two major difficulties when inserting code into binaries are

1. correctly distinguishing between code and data (especially when code is located in data, when data is located in code, or when self-modifying code is used);
2. correctly relocating the code and data after inserting instrumentation code.

In existing systems, these two difficulties can only be solved by applying a sophisticated analysis (disassembling the program into *basic blocks* and *control flow graphs*) of the binary. Hereby, assumptions have to be made about the origin of the code. Most systems can be broken by offering hand-written machine code to it. A careful analysis for large programs can take an inordinate amount of resources (both in time and space).

JiTI solves these problems (i) by creating two versions of the process: one for the data accesses and one for the code accesses and (ii) by not inserting instrumentation code in the process, but by replacing instructions by calls to instrumentation code. By cloning a process and by executing the code from an (instrumented) copy, and using the data from the other copy, we get rid of the need to distinguish between code and data. Moreover, as we don't insert instructions but merely replace individual instructions by other instructions no relocation is necessary, removing the overhead required by contemporary instrumentation tools.

Given the size α of the original process, a clone is created at address δ ($> \alpha$) (up to address $\delta + \alpha$). Since we do not insert instructions, but only replace instructions, the instrumented version of the instruction at address i will reside at address $\delta + i$.

We now have access to two copies, both containing the original code and data. However, one copy will be entirely considered as data, and the other copy will be entirely considered as code. In order to make sure that data is taken from one copy, and the code from the other copy, we will have to modify addresses that do not point to the right copy (by increasing or decreasing them with δ). It turns out that addresses used to access data are mostly absolute addresses, while, on contemporary microprocessors, code addresses are mostly relative addresses (position independent code). Since relative addresses do not require relocation when moved to another location in memory, the best choice is to fetch the code from the clone, and the data from the original copy of the process. This means that the relocation effort can be limited to the rare –especially on a SPARC processor– absolute code addresses (to make sure that the execution will never jump back to the non-instrumented version

of the process[1]) and to the rare code-relative data addresses that might be used in the code (e.g., for data that is located in the code, such as address tables).

In these cases where the relocation cannot be done *at cloning time* (e.g. for memory operations, where it is impossible to distinguish the memory operations that use relative addresses from those that use absolute addresses), it suffices to check the addresses *on-the-fly* by instrumenting the instructions that make use of them.

The instrumentation of the clone is performed before the program starts. In addition, code written during execution is instrumented on the fly. This can be used to deal with self-modifying code: store instructions are instrumented to write the data to the original and a trap instruction to the clone. If the trap instruction is ever executed, *JiTI* intercepts the trap and instruments the instruction (if necessary).

Although *JiTI* is applicable to most contemporary microprocessor architectures, porting it requires a non-trivial engineering effort. In this paper, we describe the SPARC implementation which was created to support RecPlay, a race detection tool [Rons99]. The operating system was Solaris from SUN. The choice for the SPARC was certainly not motivated by simplicity, for the following reasons.

1. The register windows makes it complicated to access the activation records of other procedure/functions.
2. The instructions in delay slots are harder to instrument.
3. Accessing and restoring the processor flags in user space is impossible.[2] Although *JiTI* itself does not alter the flags (even for comparing addresses with δ) it is possible that user provided instrumentation code does.

On the other hand, the instruction set of a SPARC processor is highly orthogonal and all instruction have the same length making the actual instrumentation phase easier than e.g. for an Intel processor.

The choice for Solaris was neutral. It could have been any operating systems featuring dynamic loadable libraries. *JiTI* does not require recompilation. It suffices to set the Solaris environment variable `LD_PRELOAD` to the name of the correct instrumentation library. This library is then loaded each time an application is started. The library contains a so-called `init` function; a function that is executed by the loader before the application is started. It is this function that will instrument the program (*instrumentation* phase) and make it ready for the *execution phase*. Hence, although *JiTI* manipulates a program at the very lowest level, it does not require kernel-level programming, or any modification to the operating system.

2.1 Instrumentation phase

In the instrumentation phase, a clone must be created and instrumented, and the entry point of the program must be set to point into the clone (`main+`δ). The instrumentation code itself consists of two parts: a part that must be present in order to guarantee the correct operation of *JiTI* (the instrumentation of some control transfer

[1]Some instructions that are known to make use of correct absolute addresses (such as returns), do not need to be instrumented.

[2]This is no longer true for the SPARC-V9 architecture, but as *JiTI* also has to run on older SPARC processors, *JiTI* does not use these instructions.

instructions, and the instrumentation of all the load/store operations[3]), and the user-defined instrumentation (all load/store operations in the case of data race detection).

A problem with the implementation for the SPARC processor was the fact that an instruction can be located in a delay slot in which case we have to be careful when replacing it with a control transfer instruction.

A second problem is that a control transfer instruction such as a function call (which is needed to call the instrumentation code) might itself have a delay slot which actually means that one instruction is replaced by a pair of instructions. In practice we have to resort to an instruction without a delay slot. On the SPARC, there are only two instructions that can be used for this: a trapping instruction (e.g. illegal opcode), and `BA,a` (branch always with annulation of the delay slot). As a branch has a much lower overhead than a trap, *JiTI* uses `BA,a` to jump to the instrumentation code. Unfortunately, the `BA,a` instruction has two problems: it does not keep track of a return address, which is needed to be able to return from the instrumentation code and the `BA,a` instruction has a limited range (22 bits or $\pm$ 8MB): it is impossible to jump from the clone to the instrumentation code in the *JiTI* library (applications are loaded at low addresses while dynamic libraries such as *JiTI* are loaded at high addresses in Solaris). These two problems were solved using *trampolines*. These are a kind of tiny subroutines with a built-in return address. Every individual `BA,a`-instruction must jump to its individual trampoline[4] which will in turn call (using a `CALL` that can reach every instruction) the instrumentation routine. The trampoline is automatically generated during the instrumentation phase. A basic trampoline is only 4 instructions long (saving the return address, and calling the instrumentation function) (see Figure 1).

Instructions in delay slots require special treatment. Since we cannot replace the instruction in the delay slot with a control transfer instruction, we replace the preceding control transfer instruction with a jump to a trampoline. The trampoline used is bigger as the return address now depends on the outcome of the control transfer instruction that is associated with the delay slot, and this requires some extra computation. Moreover, the trampoline also has to check for the possible annulation of the instrumented delay slot instruction.

Additionally, we also replace the instruction in the delay slot by a `BA,a` and a trampoline, in case the delay instruction would be the target of another control transfer instruction.

2.2 Execution phase

During the actual execution of the application, *JiTI* will jump (using the trampolines) to the instrumentation routine. The instrumentation routine always knows where it was called from (the address is placed in register `%l3` by the trampoline). At that location, there will be a `BA,a`-instruction, but by subtracting δ from this address, the original instruction is found back.

JiTI has to perform two steps in the execution phase: calculate the address used by the memory operation and execute the original operation.

[3] On a SPARC processor it is impossible to distinguish the memory operations that use relative addresses from those that use absolute addresses. Therefore, all memory operations have to be checked to make sure that they don't read from the (instrumented) clone.

[4] It is possible –for very large programs– that the trampoline is unreachable using a `BA,a` with a 22-bit offset. If this is the case, the instruction is replaced with a trapping instruction that jumps to a routine in *JiTI*, from where the actual instrumentation routine will be called.

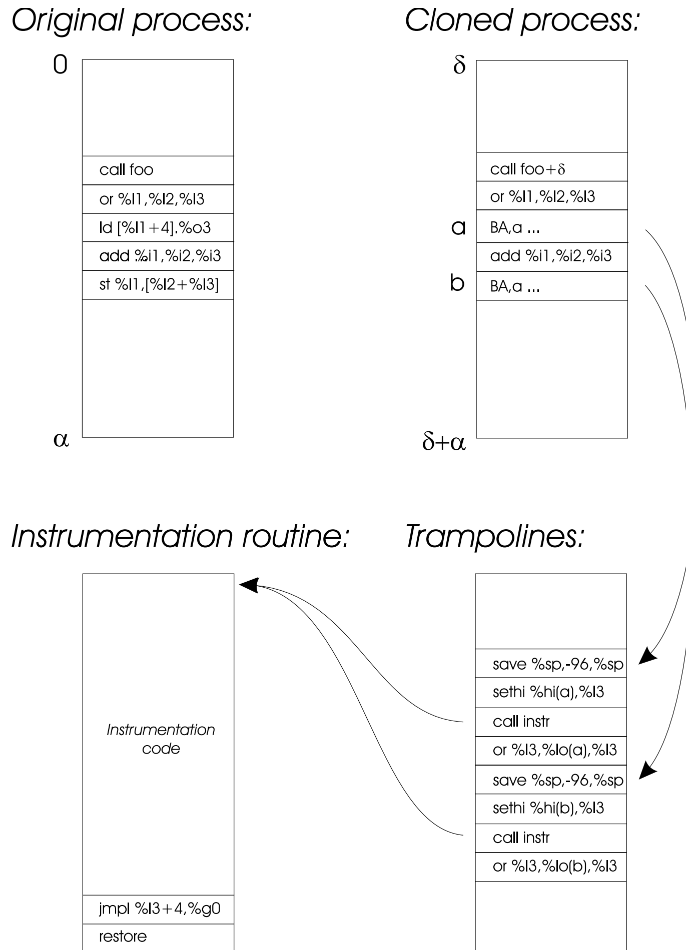

Figure 1: Using trampolines to save the return address.

Calculating the used address in the naive way, namely, decoding the instruction and explicitly processing the address calculations is quite time consuming. Therefore, we choose another approach: changing the opcode. Thanks to the orthogonal coding structure of the SPARC instruction set, the binary representation of the load/store operations, and the corresponding add-instruction that calculates the memory address used have many bits in common. By overwriting the highest 13 bits of a load/store operation with the fixed bit pattern 1000110000000, we can produce an add-instruction that will compute the source/destination address in register `%g6` (see Table 1).[5] Executing this instruction will leave the address used in this register. Of course, this means we have to save/restore `%g6`.

For executing this instruction and the original instruction a method was developed that takes advantage of a feature of the SPARC architecture that allows to place a control transfer instruction in the delay slot of another control transfer instruction. The effect of this is that the instruction at the target of the first control transfer will

[5] In case the instruction uses `%g6` in the address mode, we use another register to calculate the address. The test whether `%g6` is used is only performed once: during the instrumentation phase.

	31-30	29-25	24-19	18-14	13	12-5	4-0
`LDx [rs1+rs2],rd`	11	rd	op3	rs1	i=0	unused	rs2
`LDx [rs1+simm13], rd`	11	rd	op3	rs1	i=1	simm13	
`STx rd,[rs1+rs2]`	11	rd	op3	rs1	i=0	unused	rs2
`STx rd,[rs1+simm13]`	11	rd	op3	rs1	i=1	simm 13	
`ADD rs1,rs2,%g6`	10	00110	000000	rs1	i=0	unused	rs2
`ADD rs1,simm13, %g6`	10	00110	000000	rs1	i=1	simm13	

Table 1: Opcodes used for load and store instructions and additions. 'x' denotes the width of the memory value used, as defined by op3.

be executed as a single instruction after the second control transfer, after which the program simply continues.

The SPARC processor uses two registers (PC and nPC) to deal with control transfer instructions. The 32-bit PC contains the address of the instruction currently being executed and nPC holds the address of the next instruction to be executed (assuming a trap does not occur). Most of the time, nPC=PC+4. During the execution of a delay instruction, the nPC points to the target of the control transfer instruction, while the PC points to the delay instruction. The jump-and-link instruction `jmpl jump_address,link_register` changes PC and nPC (concurrently) as follows:

reg[link_register] $\leftarrow$ PC;
PC $\leftarrow$ nPC;
nPC $\leftarrow$ jump_address;

In order to execute the generated add-instruction, *JiTI* uses the combination that is shown in Figure 2. This method is simple and works without interpreting the instruction, without generating code, and without the need to flush the cache.

After executing the modified instruction but before executing the original instruction, the instrumentation routine must check the address that will be used by the original instruction (this address was calculated by executing the modified instruction). For memory operations, the address should point to the original process (address $< \delta$). Memory operations that use relative addresses (e.g. for accessing data in the code segment) will use addresses $> \delta$, possibly reading instrumented 'code'. Therefore, we make sure in this case that the address is smaller than δ by subtracting δ. The opposite is true for control transfer instructions that use absolute addresses (only `jmpl` on SPARC processors). We have to make sure that the address used is larger than δ, forcing the control transfer instruction to stay in the instrumented clone. Therefore, if the address used is smaller than δ, we simply add δ and jump to this address.

Store operations require special attention as it is possible that they are used to change instructions (self-modifying code). Instead of writing an instrumented version of the instruction to the clone area *JiTI* writes a trapping instruction in the clone. If the new instruction is executed, a trap will occur. *JiTI* intercepts the trap and will instrument the instruction (and possibly create a trampoline) at that moment. This way, store operations that wrote real data (the most common case) will not be instrumented, limiting the number of trampolines created.

2.3 Extra features

Due to the dynamic nature of *JiTI*, dynamic instrumentation is possible. This allows for the instrumentation of self-modifying code (see above) and for two additional types of dynamic instrumentation:

in time: it is possible to add or remove the instrumentation from the clone during the execution. This feature is used by our data race detection tool: as data races only occur in a parallel execution, there is no need to trace the memory operations that are performed during the sequential start of the program. This type of instrumentation is accomplished by starting with an uninstrumented clone that is instrumented upon the creation of a second thread (the `thread_create()` function that performs this is intercepted by *JiTI*).

in space: it is possible to limit the instrumentation to certain parts of the program. This feature is also used by our data race detector: we trace memory operations performed by the application or statically linked code, but not by dynamically linked code (that contains the Solaris synchronization operations). This approach enables us to make a distinction between data and synchronization races [Rons99]. This type of instrumentation is accomplished by forcing *JiTI* not to instrument the `PROCEDURE_LINKAGE_TABLE` code that is used to find the location of dynamically linked code.

address	instruction
1000	`instr1`
1004	`jmpl %g6,%g6`
1008	`jmpl %g6+8,%g0`
1012	`instr2`
⋮	⋮
2000	`add %l1,8,%g6`

Figure 2: A fragment of code with a control transfer instruction in the delay slot of another control transfer instruction. The instructions are executed in the following order: 1000, 1004, 1008, 2000, 1012. Here `%g6` is used to hold in succession the jump address, the return address and the result value (see Table 3)! The instruction at address 2000 is the instruction that was generated by *JiTI* to calculate the memory address, in this case for the instruction `ld [%l1+8],%l2`.

PC	nPC	%g6	instruction executed
1000	1004	2000	`instr1`
1004	1008	2000	`jmpl %g6,%g6`
1008	2000	1004	`jmpl %g6+8,%g0`
2000	1012	1004	`add %l1,8,%g6`
1012	1016	reg[%l1]+8	`instr2`

Figure 3: Executing the code that is depicted the previous table.

3 Experimental evaluation

JiTI was thoroughly evaluated using a number of small test program with code in data, data in code, self-modifying code and hand-written assembly code. One should notice that the occurrence of these constructions is not abnormal in contemporary code. E.g. in order to support C-functions that return a structure (Figure 4) the

```
typedef struct _test {int a;} test;

test foobar(void){
  test c;
  return c;
}

void main(){
  test b;
  b=foobar();
}
```

Figure 4: A C-program containing a function that returns a `struct`.

compiler places data between code in the object code[6]: the size of the structure (4 bytes) is placed between the instructions of the `main`-routine (line 10968) and this number will be checked by the `foobar()`-function to make sure that there is enough space on the stack (Figure 5). In this case, the number 4 has no meaning

```
00010958 <main>:
10958:  SAVE %sp, -104, %sp
1095c:  ADD %sp, 0x5c, %l0
10960:  CALL 10948 <foobar>
10964:  ST %l0, [ %sp + 0x40 ]
10968:  UNIMP 0x4
1096c:  LD [ %sp + 0x5c ], %o1
10970:  RET
10974:  RESTORE
```

Figure 5: The object code generated by a compiler for the code depicted in Figure 4.

as an instruction (`unimp`), but a struct with another length could cause problems for certain tools.

The occurrence of code in data and self-modifying is also not unlikely in current software. E.g. applications that are dynamically linked (almost all software on Solaris) uses a `PROCEDURE_LINKAGE_TABLE` to enable dynamic linking. This is a table in the data segment that is changed, during the execution, by the dynamic linker. Figure 6 shows, on the left, the code before and, on the right, after the dynamic loader did its work. As *JiTI* instruments code just before the execution starts, but after the dynamic loader did its work, *JiTI* will see, and instrument, the right version of the code.

[6]Using the Sun compiler or gcc on a SPARC at least.

00020a3c:	SAVE %sp,-64,%sp	SAVE %sp,-64,%sp
00020a40:	CALL ff3b2d68	CALL ff3b2d68
00020a44:	NOP	NOP
00020a6c:	SETHI 30,%g1	SETHI 30,%g1
00020a70:	BA,a 20a3c	SETHI 3fc47f,%g1
00020a74:	NOP	JMPL %g1+1016,0
00020a78:	SETHI 3c,%g1	SETHI 3c,%g1
00020a7c:	BA,a 20a3c	SETHI 3fc65a,%g1
00020a80:	NOP	JMPL %g1+8,0

Figure 6: The PROCEDURE_LINKAGE_TABLE, before and after the dynamic linker did its work.

program	no. of	instr. replaced			instr. executed	
	instr.	mem	jmpl	%	mem	jmpl
qs_sequential	211	33	2	16.6%	3 935 868	0
qs_parallel	286	37	2	13.6%	3 935 868	0
eqntott ex0.eqn	10 876	1078	19	10.1%	18 475	60
espresso dc1.in	69 344	21 938	27	31.7%	324 537	661
compress in	3 688	832	3	22.6%	4 666 683	0
uncompress in.Z	3 688	832	3	22.6%	3 272 282	0
sc < load1	43 171	7 921	23	18.4%	32 992 768	1 574
xlisp li-input.lsp	24 866	7 119	16	28.7%	33 490 004	225 873
ijpeg specmun.ppm	72 064	29 576	656	42.0%	2 612 267	939
perl primes.pl	118 258	30 016	82	25.4%	3 351 214	103 839
m88ksim < ctl.raw	53 828	14 874	50	27.7%	412 533 658	1 468 231

Table 2: Results for test programs: the total number of instructions, the number of instrumented instructions and the number of executions of these instructions.

It is nice to know that *JiTI* is able to handle hard to instrument code but the main criterion used to compare instrumentation tools is the slowdown caused by the instrumentation. Therefore, tables 2 and 3 show some results obtained for quicksort and some SPECint92 and SPECint95 programs. We used the *JiTI* implementation described above: the addresses used by all memory operations were collected. The first table shows static and dynamic information. The second column shows the number of instructions[7] in the *code* section of the program. The next two columns show the number of instructions (either memory operations or control transfer instructions) that were replaced, during the instrumenting phase, by a branch to a trampoline. Of course, this figure also includes the number of 'instructions' in the data segment that were replaced. The next column shows the percentage of instructions that were instrumented. The last two columns show the number of executed instrumented instructions. This is the number of times *JiTI* was called by the program to guarantee that the clone keeps executing correctly: fetching code from the clone and data from the original process. The huge amount of indirect jumps for *xlisp*, *perl* and *m88ksim* is caused by the fact that these programs are in fact interpreters and hence several time consuming register indirect jumps are used (especially for *m88ksim*).

[7]This number was obtained by using a disassembler to count the number of instructions.

program	execution time		setup	slowdown	
	normal	*JiTI*	time	*JiTI*	ATOM
qs_sequential	0.51	7.50	0.12	14.7	4.92
qs_parallel	0.34	4.36	0.13	12.8	N/A
eqntott ex0.eqn	0.41	0.74	0.21	1.8	5.35
espresso dc1.in	0.11	1.16	0.80	10.5	7.91
compress in	0.50	5.05	0.14	10.1	4.34
uncompress in.Z	0.44	3.92	0.14	8.9	4.12
sc < load1	10.77	85.50	0.60	7.9	3.12
xlisp li-input.lsp	2.56	54.30	0.36	21.2	7.86
ijpeg specmun.pmm	0.66	13.56	0.86	20.5	7.75
perl primes.pl	0.52	6.89	0.75	13.2	7.57
m88ksim < ctl.raw	25.28	695.28	0.71	27.5	7.35
average				11.3	5.75

Table 3: Results for test programs: the execution times: normal execution, instrumented execution and time to instrument the program.

Table 3 shows information about the execution times. The second column shows the normal execution time for the uninstrumented program. The next column shows the execution time when *JiTI* is turned on. This figure includes the *setup time*, i.e., the time to setup a clone and instrument it, and the execution time. The next column shows the setup time alone. We notice that *JiTI* introduces a small setup time (0.1 - 0.9s) per program run, only a small fraction of the total execution time. Hence, there is no need to statically instrument the program. It is instead easier to redo the instrumentation before every run. The last column shows the slowdown which is on the average limited to 11.3. This is reasonable given the fact that every load/store instruction is replaced by a jump to a trampoline, which in turn calls the instrumentation routine.

The last column of this table is the slowdown we obtained with ATOM, a widely used instrumentation tool for Alpha machines. ATOM took between 4 and 10 seconds to produce the instrumented program (on a much faster machine). This should be compared with the time *JiTI* needs to instrument the program (0.1-0.9s). The average slowdown using ATOM is about 2 times better than using *JiTI*; this is caused by the fact that Alpha code is easier to instrument as there are no delayed branches or register windows.

4 Related work

There are not that many systems that allow to trace at the machine instruction level. Probably the best known static binary instrumentation tool for SPARC based machines is EEL [Laru96]. EEL inserts so-called *snippets*, containing new code, into an application. There are some places where snippets cannot be inserted, e.g. after a control transfer instruction. EEL cannot deal with self-modifying code and unrestricted indirect jumps and has problems with hand-written assembly code [Arpa96].

A dynamic binary instrumentation tool is Paradyn and its derivated DynInst API [Mill95, Holl97]. The tool runs on a number of processor architectures (including SPARC) and was developed for performance measurements. The system allows

instrumentation (in the form of *code patches*) to be added ('spliced') at procedure entries, procedure exits and individual call statements. The instrumentation routine can operate on counters, timers, constants, parameters to a procedure or a procedure return value. Therefore, the instrumentation of a single instruction or the detection of the address used in a memory operation is not possible. As Paradyn does not insert instructions but replaces instructions by a call to a routine, the target of a control transfer instruction will not change. Self-modifying code cannot be dealt with.

A novel and dynamic instrumentation tool for instrumenting kernels is described in [Tamc99a, Tamc99b]. Although the tool is dynamic, it performs the same analysis as static tools to create an interprocedural control-flow graph of basic blocks and finding live registers for each basic block. As such the tool has the same problems as static tools: data in code, code in data and handwritten and self-modifying code. However, relocation is not needed as the tool replaces instructions with a branch to *springboards* (the equivalent of trampolines in *JiTI*) to jump to the actual instrumentation code. Unfortunately, the tool deals in an incorrect way with delayed branches: the delayed branch and the delayed instruction are both moved to the springboard and executed over there. This is incorrect if the delayed instruction uses the link register used by the control transfer instruction: as the instructions are moved, the return address saved in the link register is not the original address. Moreover, the delay instruction is *not* instrumented. This causes problems if the delay instruction is the target of a control transfer instruction (see section 2.1).

5 Conclusions

In this paper, we have described *JiTI*, a program instrumentation technique that is able to correctly instrument hard to instrument features such as data in code, code in data and self-modifying code. *JiTI* is implemented as a dynamic library, allowing it to be applied to existing applications. Hence, the instrumentation is a property of an execution, and not of the program itself. *JiTI* has been implemented for the SPARC architecture, and is currently used as basic infrastructure for our RecPlay data race detector.

Acknowledgements

The authors would like to thank Jan Van Campenhout for his support and advice. This work was supported in part by the *Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)* under grant number 174WT800. Koen De Bosschere is research associate with the Fund for Scientific Research — Flanders.

References

[Arpa96] R. Arpaci and M. Fähndrich. revEELing Solaris. EECS Department, University of California, Berkeley, May 1996.

[Ball92] T. Ball and R. Larus. Optimally Profiling and Tracing Programs. Conference Record of the 19th ACM Symposium on Principles of Programming Languages, 59–70, 1992.

[DB96] K. De Bosschere and S. Debray. `alto`: a Link-Time Optimizer for the DEC Alpha. Technical Report TR 96-15, Computer Science Department, University of Arizona, 1996.

[Holl97] J. Hollingsworth, B. Miller, M. Goncales, O. Naim, Z. Xu, and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proceedings of Intl. Conference on Parallel Architectures and Compilation Techniques*, San Fransisco, July 1997.

[Laru96] J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. SIGPLAN Conference on Programming Language Design and Implementation, June 1996.

[Lee] D. Lee, T. Romer, G. Voelker, A. Wolman, W. Wong, B. Chen, B. Bershad, and H. Levy. Instrumentation and Optimization of WIN32/Intel Executables. URL= `http://etch.cs.washington.edu/`.

[Mill95] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Tools. *IEEE Computer*, 28(11):37–44, November 1995. Special issue on performance evaluation tools for parallel and distributed computer systems.

[Rons99] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[Sriv94] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. Research report 94/2, Digital-WRL, 1994.

[Tamc99a] A. Tamches and B. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Third Symposium on Operating Systems Design and Implementation*, 117–130, New Orleans, February 1999.

[Tamc99b] A. Tamches and B. Miller. Using Dynamic Kernel Instrumentation for Kernel and Application Tuning. *Parallel Processing Letters*, 1999.

[Xu99] Z. Xu, B. Miller, and O. Naim. Dynamic Instrumentation of Threaded Applications. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Georgia, May 1999.

[Zand99] V. Zandy, B. Miller, and M. Livny. Process Hijacking. In *Proceedings of the 8th IEEE Int'l Symposium on High Performance Distributed Computing*, Redondo Beach, California, August 1999.