Combining Light Static Code Annotation and Instruction-Set Emulation for Flexible and Efficient On-the-fly Simulation

Thierry Lafage, André Seznec

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Combining Light Static Code Annotation and Instruction-Set Emulation for Flexible and Efficient On-the-fly Simulation

Thierry Lafage, André Seznec

N°3821

Décembre 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3821--FR+ENG

INRIA
RENNES

Combining Light Static Code Annotation and Instruction-Set Emulation for Flexible and Efficient On-the-fly Simulation

Thierry Lafage, André Seznec

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n ° 3821 — Décembre 1999 — 25 pages

Abstract: Computer architects rely on on-the-fly simulation for designing microprocessors. However, these simulations cannot use large workloads consuming several CPU hours since, using current software approaches, even skipping instructions, is very time consuming.

This paper proposes a new cost effective trace collection and on-the-fly simulation approach. The original application code is lightly annotated to provide a fast (direct) execution mode. An embedded instruction-set emulator enables trace collection or on-the-fly simulations. At run time, dynamic switches are enabled from the fast mode to the emulation mode by the annotation code, and vice-versa.

The instrumentation tool, **calvin2**, and the instruction-set emulator, DICE, presented in this paper, exhibit low execution slowdowns (from 1.01 to 3.09) on the SPEC95 benchmarks running in fast mode. Such low execution slowdowns make it possible to use samples spread over a long running application for simulations. We compare our approach with the dynamic translation approach used in Shade. While our DICE emulator is not optimized for raw performance, our system will outperform dynamic translation provided that only a small part of the application execution is effectively simulated.

Key-words: software tracing, static code annotation method, dynamic code emulation, micro-architecture simulation, trace-driven simulation, on-the-fly simulation, trace-collection platform.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Association d'une instrumentation de code légère à l'émulation du jeu d'instructions pour des simulations "au vol" flexibles et efficaces

Résumé : La conception de microprocesseurs nécessite de plus en plus l'utilisation de simulations "au vol". Cependant, les techniques logicielles actuelles ne permettent pas de simuler de gros programmes s'exécutant plusieurs heures en un temps raisonnable, même en ne simulant qu'une partie de ces programmes.

Ce papier propose une méthode de collecte de traces et de simulations "au vol" efficace. Le code original est légèrement instrumenté et donne un mode "rapide" d'exécution du programme. Un émulateur de jeu d'instructions embarqué dans le programme permet le traçage ou la simulation. À l'exécution, le code d'instrumentation permet de passer du mode "rapide" au mode émulé et vice-versa.

L'outil d'instrumentation, **calvin2**, et l'émulateur de jeu d'instruction, DICE, présentés dans ce papier ralentissent peu les applications tracées (SPEC95) en mode "rapide" (de 1.01 à 3.09). De tels ralentissements permettent d'envisager de ne tracer/simuler que quelques échantillons répartis sur une longue exécution. Nous comparons cette approche avec la traduction dynamique utilisée dans l'outil Shade. Bien que notre émulateur DICE ne soit pas optimisé, notre système s'avère plus rapide que la traduction dynamique dans les cas où seulement une petite partie de l'exécution est tracée/simulée.

Mots-clé : traçage logiciel, instrumentation de code, émulation de code, simulation de micro-architecture, simulation dirigée par la trace, simulation "au vol", plateforme de collecte de traces

1 Introduction

On-the-fly simulation is widely used to evaluate microprocessor architecture and memory system performance. The involved simulations require realistic program executions. The addresses of instructions executed and/or data referenced by a program (for cache simulations), or registers or memory values (to simulate value prediction for instance) must be accessed by the simulator.

The trace-collection process is time consuming: even for simply collecting addresses for memory hierarchy simulations, current general-purpose trace-collection tools slow down traced program execution by a factor of, at least, 10 [24] [1]. For studies like value prediction [5, 16], more information must be collected, so execution slowdowns will be higher (over 100). In addition, micro-architecture (or memory system) simulation induces much higher slowdowns with respect to the workload execution time (in the 1,000–10,000 range [4, 3]).

To reduce simulation times, trace sampling, as suggested by [14, 19] for cache simulations, may be used. For long running workloads, one cannot reasonably expect to use the complete trace to extract samples because storing the full trace would need hundreds of giga bytes of disk and would take hours. For such workloads, on-the-fly simulation is the only acceptable solution. However, current trace-collection tools are not really suited to such techniques because they are made with the assumption that programs are to be traced entirely: at best, they provide a "skip mode" to position the simulation at a starting point. However the "skip mode" still exhibits a high execution overhead between trace samples [2]. Thus, using current tracing tools, trace sampling does not make it possible to simulate on-the-fly samples spread over a long application.

As a consequence, apart very specific studies like cache behavior analysis of the whole SPEC92 benchmark suite [10] (which used months of CPU time), micro-architecture studies are performed using the first instructions (maybe some billions) of an application, maybe skipping the first billions to avoid the initialization phase (e.g. see [6]). This solution can be viewed as trace sampling with one large sample taken after the beginning of the program execution. As said earlier, large applications with long initialization phases cannot be traced this way.

For these reasons, computer architects relying on trace or execution-driven simulation do not use realistic (large) workloads consuming several CPU hours.

This paper presents the first implementation of a new on-the-fly simulation approach. This approach takes advantage of both static code annotation and instruction-set emulation in order to provide traced programs with two execution modes: a fast (direct) execution mode and an emulation mode. A light code annotation is used for the fast mode: code is added to the traced program only to enable dynamic control transfers to an embedded host instruction-set emulator. When the embedded emulator is granted the program control, user-provided simulation routines are executed. At run time, dynamic switches are allowed

[1] This is, however, not true for very specific studies such as presented in [15].

[2] This is the case of *time sampling*. In the case of *set sampling*, the trace may be entirely collected before the samples are chosen.

between the fast mode and the emulation mode, and vice-versa. Dynamic switches are triggered by what we call *switching events*.

The fast execution mode is expected to exhibit a very low execution overhead over the native program execution, and therefore makes it possible to fast forward billions of instructions in a few seconds. In addition, the instruction-set emulator is flexible enough to allow the user to easily implement different micro-architecture or memory hierarchy simulators. Instruction-set emulation makes it possible to emulate the entire *user* activity, including dynamically linked code and dynamically compiled code.

Consequently, our approach is well suited to simulate samples spread over long running applications since most of the native instructions are expected to execute in fast mode.

In the next section, we present related works about code emulation used for tracing or simulations. Section 3 details the approach of combining light static code annotation and instruction-set emulation. Section 4 presents our static code annotation tool: **calvin2**. Section 5 presents DICE, our instruction-set emulator. Section 6 evaluates the performance of **calvin2** + DICE on the SPEC95 benchmark suite, and compares it to Shade [7]. Section 7 summarizes this study and presents our directions for future development.

2 Related Work

Static annotation has been used to trace applications for about ten years. While these tools generally exhibit lower overheads than code emulation [24], they cannot trace dynamically compiled or dynamically linked code. Also, they do not enable execution-driven simulation as required to simulate current (and future) speculative out-of-order execution superscalar processors.

As our approach is intended to allow both trace-driven and execution-driven simulations, we will focus here on instruction-set emulation tools.

Shade [7] is an instruction-set emulator based on dynamic translation of emulated instructions. On the first occurrence of a basic block in an application, Shade translates it in a sequence of native instructions. In this sequence, instrumentation code for trace recording or simulation is embedded. On further occurrences of the same basic block, the emulation sequence is directly invoked by the emulator. Trace recording is made by a user defined analysis routine. Slowdowns from 9 to 14 were reported using Shade for address traces instrumentation.

To our knowledge, Shade is the fastest instruction-set simulator for user-level activity. Also, as an instruction-set simulator, it provides a very flexible user interface. However, switching from program execution to program simulation is not possible with Shade. On-the-fly simulation sampling is only possible if it is entirely managed by the user analysis routine (called by the instrumentation code added by Shade, during the translation). As a result, program parts which are not simulated are still time consuming. In Section 6, we will show that while our DICE instruction-set emulator is not as fast as Shade, our approach combining **calvin2** and DICE is much more efficient.

SimOS is a complete simulation environment capable of modeling an entire computer system, including operating system and all application programs [22]. SimOS associated with Embra [29] provides a less detailed-execution mode to position the target workloads. It is used on code sections the user is not interested in tracing (e.g. the operating system boot or the beginning of a user application). This *positioning mode* relies on the same principle of dynamic translation as Shade and is the fatest mode of SimOS. Other slower and more detailed simulation modes are provided: *rough characterization mode,* and *accurate mode.* SimOS makes it possible to dynamically switch from a simulation mode to a more detailed one. Also, to handle the large amount of low level-data generated by the hardware simulation models, SimOS provides flexible annotation and event classification mechanisms that map the data back to concepts meaningful to the user [21].

SimOS is a powerful tool and one of the very first to provide different execution modes. However, as Shade, SimOS only relies on total program simulation. To model a uniprocessor, SimOS execution slowdowns in *positioning mode* range from 20 to 30 [11]: these are the lowest slowdowns possible.

MIME [23] is a simulation and tracing tool designed to support trace-driven simulations. MIME is an emulation engine for the HP Precision Architecture (v1.1). MIME can switch between emulation of a program and its execution on the actual hardware at definite intervals of time using timer interrupts.

Our approach is very close to the MIME approach as we also rely on real execution and emulation. However, dynamic switches between direct execution of the program and its emulation in MIME are dictated by timer interrupts: this makes MIME unable to reproduce several identical emulation sequences.

3 Light Static Code Annotation and Instruction-Set Emulation

To trace programs or to perform on-the-fly simulations, static code annotation is generally a more efficient technique than instruction-set emulation [15, 24]. However, instruction-set emulation is generally a much more flexible approach: 1) it makes it possible to implement different tracing/simulation strategies without the need to (re)instrument the target programs and 2) dynamically linked code and dynamically compiled code are traced and simulated easily.

First, our approach takes advantage of the efficiency of the static code annotation technique: a light instrumentation provides target programs with a fast (direct) execution mode which is used to rapidly position the execution in interesting tracing states.

On the other hand, an instruction-set emulator is used to actually trace the target program or enable on-the-fly simulations. This emulator is embedded in the target program, and is then able to take the control during the execution.

At run time, the program switches from fast mode to emulation mode whenever *switching events* happen. The inserted annotation code only tests whether a switching event has

occured, and when this is the case, it gives the control to the emulator. Note that, mode switching is made deterministic since the annotation code drives it. Switching back from emulation mode to fast mode is managed by the emulator and is possible at any moment.

The fast execution mode is expected to incur a very low execution overhead compared to the emulation mode. Accordingly, this approach is well suited to simulate samples spread over long running applications, since most of the native instructions will execute in fast mode (low sampling ratio).

4 Light Static Code Annotation with calvin2

calvin2 is a static code annotation tool derived from **calvin** [13]. **calvin2** lightly instruments the target programs by inserting *checkpoints*. **calvin2** has been built using SALTO [20], a system for assembly-language transformation and optimization, and can instrument SPARC assembly code.

The checkpoint code sequence consists in a few instructions (about 10) which checks whether the control has to be given to DICE, the emulator. We call the direct execution of the instrumented application *fast execution mode*, as we expect this mode to exhibit a performance close to the original code performance. When a checkpoint gives the control to DICE, the application runs in *emulation mode*. Switching from fast mode to emulation mode is triggered by a *switching event*.

In the next subsection, the number and position of the checkpoints among the program code are discussed. Then, Subsection 4.2 details several switching events types and the associated content of the checkpoints.

4.1 Checkpoint Code Insertion

The number of inserted checkpoints directly determines the execution overhead in fast mode (they have a negligible influence in emulation mode as DICE locates checkpoint code and does not emulate it). So checkpoints must not be too numerous. In contrast, their number and distribution among the code executed determines the dynamic accuracy of mode switching (fast mode to emulation mode). For instance if the checkpoint distribution was uniform every 500 dynamic instructions, we could choose to enter emulation mode at the N^{th} instruction $\pm$ 250 (ideal accuracy) at a cost of only around 2 % performance (10 instructions added each 500 instructions).

Unfortunately, checkpoints have to be inserted at code generation. The difficulty is to find a static checkpoint layout which is interesting for as many programs as possible, and for as many usages as possible (i.e. whatever switching events are) while still incurring only a small execution overhead.

4.1.1 Dynamic Distribution of Checkpoints

We intend to fit a wide range of applications. Then we decided *a priori* to put checkpoint annotation code near each procedure call because procedure calls are relatively rare, but are not so uncommon. However, scientific programs may execute millions or even billions of instructions within a single procedure call. As these programs are *a priori* composed of many loops, we have decided to insert checkpoints at each procedure call and inside each path of a loop [3].

We gathered statistics on the checkpoint distributions in the SPEC95 benchmarks. DICE was run on the instrumented programs until completion, using the *ref* input data set. We gathered the overall count of instructions and checkpoints executed [4] as well as the distances between two consecutive checkpoints (number of instruction executed between them).

The results are summarized in Table 1 (σ is the standard deviation). For each program, the last column provides a rough evaluation of the execution slowdown in fast mode. An execution slowdown is given by $Slowdown = \frac{instrumented\_exec\_time}{original\_exec\_time}$. Code inserted for a checkpoint is about 10 instructions. Assuming that CPI [5] performance is equivalent for both the instrumented program and the original one, the slowdown can be approximated by:

$$Est\_Slowdown = 1 + \frac{10 \times nb\_checkpoints}{original\_nb\_instructions}$$

Table 1 shows that the number of checkpoints inserted in the SPEC95 programs is quite reasonable (4.57 % on average for CINT95 programs, and 2.50 % on average for CFP95 programs) and allows us to expect quite low execution slowdowns (between 1.02 and 1.70). Effective slowdown is discussed in Section 6.

For each workload, the regularity in the checkpoint distribution among dynamic instructions is represented by the inverse of the standard deviation (σ). Except fpppp [6], the applications exhibit standard deviation values that are lower than 100 instructions.

Moreover, even if the maximum dynamic distance values are high compared to the mean values (tens of thousands vs. 14.2–82.8), we checked [7] that only a few distances (< 10 %) have very high values. In particular, 90+ % of the distances are less than 200 for all the programs, and 90+ % of the distances are less than 100 for all the programs but two. Figure 1 shows

[3] We used the classical algorithm from Aho, Sethi and Ullman [2] to locate loops inside the assembly code. Basic blocks are numbered in depth first. Backward edges are edges whose head is lower than the tail, and indicate the presence of a loop.

[4] As we used DICE, the data collected do not only reflect instrumented program code execution, but also dynamically linked libraries, as well as the dynamic linker code execution, even though not instrumented.

[5] Cycles Per Instruction.

[6] From `http://www.specbench.org` this application spends 35.5 % of the overall time executing a procedure composed of one very large basic block (6000+ instructions). We assume this feature to be very specific to this application and we did not change our instrumentation tool to enable checkpoint insertion inside large basic blocks. Instead, checkpoints are inserted at the beginning of the chosen basic blocks.

[7] Because of its very specific structure, we exclude fpppp from these comments.

	Million instr. executed	**Million check-points** executed (ratio †)	**Distances between checkpoints** min.	mean	max.	σ	**Est.** slowdown
compress95	51011	3335 (6.54 %)	3	15.3	3608	14.5	1.65
gcc *	976	43 (4.42 %)	1	22.6	42387	88.2	1.44
go	32149	1315 (4.09 %)	2	24.4	3674	17.6	1.41
ijpeg	33940	863 (2.55 %)	2	39.3	18860	35.8	1.25
li	62254	4369 (7.02 %)	2	14.2	3340	13.0	1.70
m88	79869	5050 (6.32 %)	2	15.8	16468	14.6	1.63
perl	27714	707 (2.55 %)	2	39.2	41246	12.9	1.26
vortex	69880	2166 (3.10 %)	4	32.3	117739	24.2	1.31
applu	83692	2779 (3.32 %)	5	30.1	18007	29.0	1.33
apsi	45181	1278 (2.83 %)	5	35.3	178493	32.7	1.28
fpppp	172570	312 (0.18 %)	5	552.4	8797	551.3	1.02
hydro2d	63743	2696 (4.23 %)	3	23.6	91933	22.1	1.42
mgrid	113831	1374 (1.21 %)	6	82.8	19230	82.4	1.12
su2cor	37121	815 (2.20 %)	3	45.5	32966	43.8	1.22
swim	42539	712 (1.68 %)	5	59.7	28919	57.5	1.17
tomcatv	59361	981 (1.65 %)	11	60.5	24884	57.8	1.17
turb3d	157558	5500 (3.49 %)	4	28.6	29073	28.0	1.35
wave5	38926	1636 (4.20 %)	4	23.8	25401	20.3	1.42

†. 100 × (number of checkpoints/number of instructions).

*. The information supplied here concerns only one of the 55 executions performed by the benchmark with the *ref* input data set. Other executions provide quite similar results.

Table 1: Statistics about checkpoints at procedure calls and within each loop running the SPEC95 benchmarks.

the distributions of the checkpoints for the CINT95 go program and the CFP95 hydro2d program. Checkpoint distributions for the other SPEC95 programs are quite similar.

Therefore, for all the programs but fpppp, given a dynamic switching event, there is a very good probability that the execution mode switches actually happen within less than 100 instructions. So, we think that our choice of checkpoints is quite acceptable.

4.2 Switching Events

We call *switching event*, the event that, during the execution in fast mode, makes the next executed checkpoint pass control to DICE. Switching back from the emulation mode to the

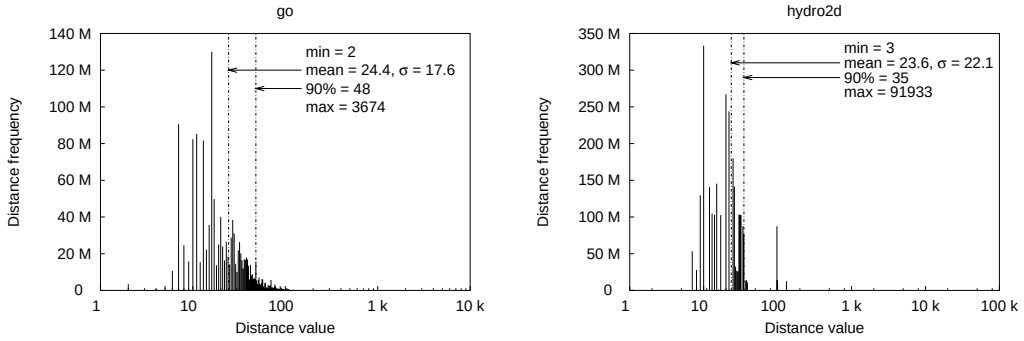

Figure 1: SPEC95 go and hydro2d checkpoint distributions.

fast mode is determined by the simulation user (e.g. a given number of instructions emulated or some point reached in the execution).

We describe below the four different types of switching event we have implemented so far.

4.2.1 "inner" Switching Event Type

The "inner" type is the first and simplest switching event type implemented: control is passed to DICE when a given number of checkpoints have been executed. This type of event enables to sample the simulation over a complete run of the target program: for instance, simulating 50 000 000 instructions every billion checkpoints executed.

When the target program is run, a global counter is first set to a user-defined value; then, each checkpoint decrements the counter by one. When the counter reaches zero, the control is transfered to DICE. Note that when the control switches back from the emulator to the lightly instrumented code, the counter may or may not be set to the same value.

4.2.2 "shm" Switching Event Type

The "shm" (shared memory) event type is a mean of synchronization to enter (resp. exit) emulation mode, on a multiprocess workload.

It is implemented as follows: the execution mode (emulation mode or fast mode) of the target program processes are guided by a shared variable value (boolean). Checkpoint code in all the target processes reads the shared variable and, switches to the emulation mode when it is 1 (true). An external process controls the execution mode by writing the shared variable.

4.2.3 "ra" Switching Event Type

The "ra" (routine address) switching event type is used to rapidly fast forward the execution of the target program to a given routine before the emulation mode is entered and the simulation begins.

The checkpoints first compare their own address with a global variable. If the values are equal, then, like with "inner" type, a global counter is decremented by one. When the counter is zero, the control is passed to DICE (emulation mode). Also, we provide a perl script that makes it possible to relate checkpoint addresses with program routine symbols in order to correctly set up the "sampled execution".

During the execution, several "tracing sessions" may take place: after the first session, e.g. once DICE has emulated for the first time the user defined number of instructions, the control is given back to the program, and a new session is set up. The target programs are initialized with a threshold value for the counter and a checkpoint address, for the first session. For other sessions, new threshold values, and new checkpoint addresses are set up at the end of the previous sessions.

Note that using this switching event type, there is no real need to put additional checkpoints inside each loop, as suggested by section 4.1, because the checkpoints are related only with the program procedure names.

4.2.4 "hc" Switching Event Type

This switching event type may be used to do statistical trace/on-the-fly simulation sampling. To this aim, we use the performance counters available on recent microprocessors [8] ("hc" stands for hardware counters). The checkpoints compare the value of one of the counters with a threshold value (user configured). When the counter value has reached the threshold, the execution switches to emulation mode.

We made an implementation with one of the hardware counters configured to accumulate executed instructions. As we wanted the mode switch to happen after a given number of executed *native* instructions, the checkpoint code removes from the total instruction count, the number of instructions executed in all previous checkpoints. Here also, several tracing sessions can be done.

5 DICE: A Dynamic Inner Code Emulator

DICE stands for Dynamic Inner Code Emulator. DICE emulates SPARC V9 instruction-set architecture (ISA) code. DICE is a piece of C and assembly code (archive library) and is embedded in (linked with) the target application. As such, it can receive the control,

[8] This necessitates a special operating system kernel support for the processor performance counters in such a way that performance counters accumulate a per-process activity. We implemented this support on an UltraSPARC running Linux (kernel 2.2.10, RedHat 6.0 distribution). In this implementation, the performance counters are user-readable without any system call: this makes performance counter reads really fast.

and return to direct execution at any moment during the execution by saving/restoring the host processor state. DICE works with programs instrumented by **calvin2**: the inserted checkpoints are used to give control to it.

DICE enables simulation by calling user-defined analysis routines between each instruction emulated. Analysis routines have direct access to all information in the target program state, including complete memory state, and register values.

The following subsections details DICE internals: the emulation core, the processor modeled, the resulting executable memory image, the operating system interface, and the user interface.

5.1 Emulation Core

DICE emulation core is made of the traditional fetch-decode-interpret loop shown in Fig 2. Each instruction is taken in the program text segment, decoded and interpreted. Trace collection or on-the-fly simulation is allowed by calling specific user-provided routines at each iteration of the main emulator loop (c.f. 5.5).

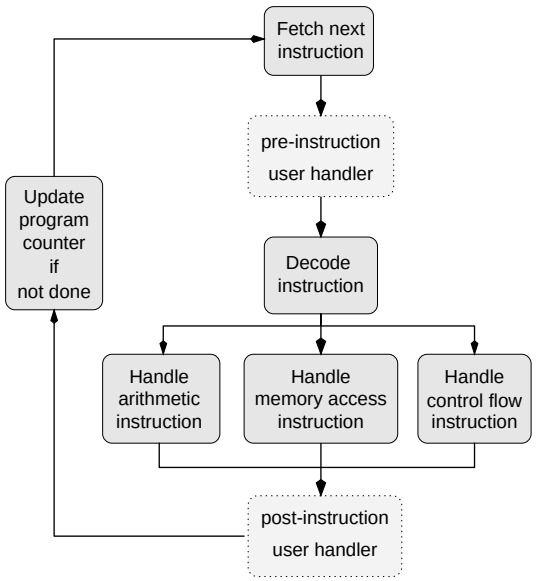

Figure 2: DICE main processing loop.

5.2 Processor Model

DICE can emulate SPARC V9 ISA code [28] and models the resources of an UltraSPARC processor [27]. These resources are used to keep in memory the state of the target program; they are made of a memory copy of all the SPARC V9 non-privileged registers: general-purpose registers, floating-point registers, and control register (PC, nPC, ...).

DICE handles most of the SPARC V9 instructions[9], as well as most UltraSPARC instructions which are part of the SPARC V9 implementation dependent instructions, like the Visual Instruction Set (VIS)[10].

5.3 Executable Memory Image

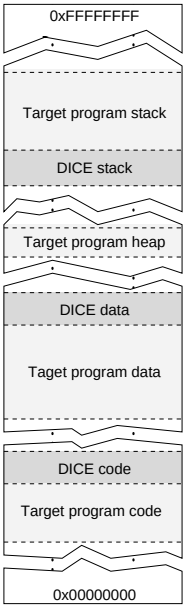

Figure 3: Target program and DICE code memory mapping.

DICE code is an archive library which is statically linked with the target program. In other words, DICE is part of the same address space as the target program.

Figure 3 shows a memory mapping of the target program and DICE code (dynamically linked library code and data are not represented). Code and data memory sections come from the executable file, and are statically mapped into memory at the beginning of the execution. In contrast, stack and heap are memory sections that dynamically evolve during the execution.

From Figure 3, we can notice that DICE does not use heap memory at all. This avoids perturbing the heap memory management, usually done by standard C libraries; the program heap data addresses are also kept the same as in the original program. In other respects, DICE does not use non-reentrant library code. This rule also applies to user tracing routines or simulation code which is embedded in the same manner (see 5.5).

Furthermore, Figure 3 shows both layouts for DICE and the target program memory stacks. During the execution, DICE maintains this layout in order to keep both DICE view and the target program view the same. This allows DICE to: 1) take control or give back control to the program without deeply modifying the stack and 2) provide realistic program stack addresses to the tracing/simulation routines with no need for original address recomputation.

The SPARC architecture features register windows. Programming conventions are that the register `%i6` (also known as `%fp`, *frame pointer*) and the register `%o6` (also known as `%sp`, *stack pointer*) define a stack frame. Figure 4 shows how DICE manages stack frames to emulate SAVE and RESTORE instructions (DICE only use two stack frames when handling these instructions). On a SAVE instruction, the target program acquires a new register window. DICE returns to the program register window and stack frame, acquires for the

[9]There are some rarely used instructions (i.e. not generated by compilers and perhaps not used in standard C libraries) DICE does not handle like: quad precision floating-point instructions, branches on floating-point condition codes with prediction, conditional move with floating-point condition codes, and compare and swap from alternate address space. Also, DICE does not handle privileged instructions as there is no need for supporting them.

[10]The implementation dependent instructions that are not handled by DICE are those few which are not even known by the assembler we use: GNU as.

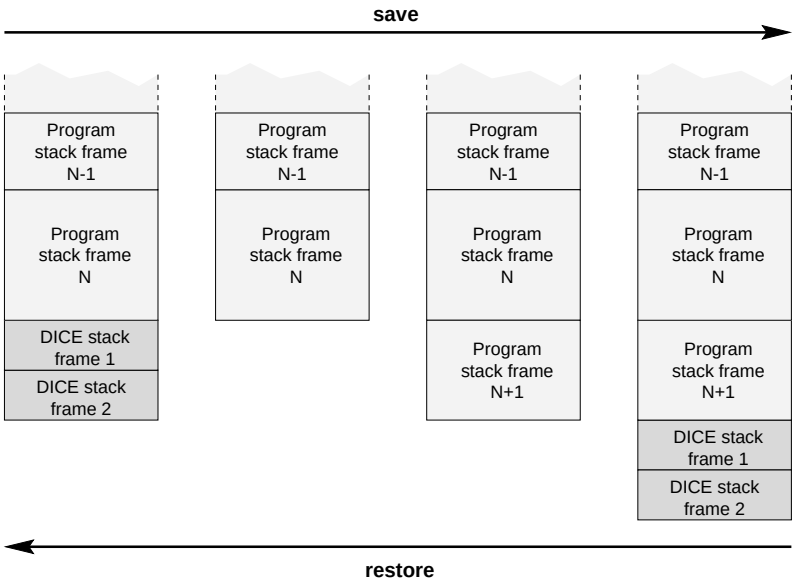

Figure 4: DICE stack frame management.

program a new register window and then resumes its own execution recreating its own stack frame and register window. The RESTORE instruction is managed the same way.

5.4 Operating System Interface

At low level, an operating system call is implemented by a software trap. On SPARC machines, arguments to the system call are passed through registers, and a trap instruction is issued. Then, the processor gives control to operating system code, in another address space. The operating system executes the routine it has been called for, then returns from the system call: the processor gives control to the user part of the process.

There is no way for DICE to follow the execution into the operating system, after a trap instruction happened. DICE trap instruction handler dumps the part of the hardware processor state necessary to pass arguments, issues the trap instruction, then saves the new processor state. Figure 5 shows a simplified version of the assembly code used.

Note that some system calls are not handled the same way: for example, `exit` and `exec` do not return.

Signals If the target program sets up a signal handler while executing in fast mode, then DICE cannot be aware of it. If a handled signal is received while DICE has the control of

```
! Load %g1-7: glob. regs
ldx [R+0x8],%g1
ldx [R+0x10],%g2
ldx [R+0x18],%g3
ldx [R+0x20],%g4
ldx [R+0x28],%g5
ldx [R+0x30],%g6
ldx [R+0x38],%g7

! Load %o0-4: arguments
ldx [R+0x40],%o0
ldx [R+0x48],%o1
```

```
ldx [R+0x50],%o2
ldx [R+0x58],%o3
ldx [R+0x60],%o4
ldx [R+0x68],%o5

! Load CCR: cond. codes
wr CCR,%g0,%ccr

! Do the trap
ta 0x10
```

```
! Get back %g1-g7
stx %g1,[R+0x8]
stx %g2,[R+0x10]
stx %g3,[R+0x18]
stx %g4,[R+0x20]
stx %g5,[R+0x28]
stx %g6,[R+0x30]
stx %g7,[R+0x38]

! Get ret. val. in %o0
stx %o0,[R+0x40]

! Get back CCR
rd %ccr,CCR
```

Figure 5: Assembly code to manage system calls.

the program, then the signal handler is called by the operating system, and DICE looses the control; note that when the hanlder returns, DICE gets the control back.

An efficient signal management would necessitate the modification of **calvin2**, or the signal management routines of the C library, to, at least, make it possible to record signals for which default behavior is modified to allow DICE to know about them. This would only be necessary to trace/simulate the programs which make intensive use of signals in non-emergency code ("normal" behavior).

5.5 User Interface

DICE provides an interface for allowing users to access dynamic information in order to trace/simulate the target programs. Various levels of detail are available and are configured at compilation time by defining (or not) preprocessing macros.

DICE user interface is implemented by some global variables and a few function declarations. The functions (user analysis routines) are to be defined by the user, and are called by DICE under well defined circumstances. Also, depending on DICE configuration, some of these functions may or may not be called.

5.5.1 Accessing Host Processor Resources

Each user analysis routine can access the host processor resources (general-purpose registers, floating-point registers and control registers) through its memory model (see 5.2), as they are defined as global variables. These variables can be read, but they are not expected to be modified by non-DICE code, and any attempt to modify one of them, although possible, will modify the behavior of the target program and alter the trace or the simulation.

5.5.2 Trace Information Detail

Some preprocessing macros may be defined before DICE compilation to enable either specific analysis routine calls, either some dynamic information to be catched by DICE, at run time. Here are each of these macros and what they enable:

- `BEFORE_INST`: call the user analysis routine `before_inst_trace()` before each instruction emulation with the program counter (PC) passed as a parameter, as well as minimal information about the type of the instruction (common, annulled, checkpoint instruction, ...).
- `SYSCALL_TRACE`: call the user analysis routine `syscall_trace()` at each system call (the system call number is passed as a parameter).
- `CHKPTS_TRACE`: call the user analysis routine `checkpoint_trace()` at each checkpoint.
- `TRACE`: tracing is enabled. The user analysis routine `after_inst_trace()` is called after each instruction emulation, and minimal information is passed to it: PC, instruction opcode, and minimal information about the instruction type (common, annulled, checkpoint instruction, ...).
- `V_TRACE`: verbose tracing. The same as `TRACE` with additional information: additional, more detailed, instruction type (memory read/write, arithmetic, floating point, ... instruction), target address (for CTIs or memory references), size of the data referenced [11].
- `VV_TRACE`: very verbose tracing. The same as `V_TRACE` with additional information: additional, accurate, instruction type (at opcode-level as defined by the SPARC V9 ISA [28]), contents of source registers before execution and destination registers after execution.
- `VVV_TRACE`: very, very verbose tracing. The same as `VV_TRACE` with additional information: indices of source and destination registers in the concerned host register set (general or floating-point).

5.5.3 Original Address Recomputation

For most simulations, instruction addresses are required. However, because of checkpoint insertion, **calvin2** changes the addresses of the native instructions in the program [12]. In order to provide the user with realistic information, we recompute the original instruction address in the application. This address recomputation is made possible by some static

[11] For memory references only.

[12] If the current instruction comes from a dynamically linked routine, then its address can be used as-is by the user analysis routine because, generally, dynamically linked routines are mapped at the same place as in the original program.

information gathered by **calvin2**, while instrumenting a program, and made available to the user analysis routines.

We can notice from Figure 3 that the target program data segment may not be mapped at the same place as the original program data segment because of the target program instruction segment dilation (added checkpoints) and DICE instruction segment mapping. However, this target program data segment is still mapped as a whole. Consequently, traced addresses coming from this segment may not be exactly the same as original data addresses, but the gap between two data in the segment is still retained. As a result, we do not recomputed addresses in the target program data segment.

5.5.4 Miscellaneous Constraints and Features

Library routines Like DICE code, user analysis routines are embedded in the traced program. As such they must observe certain constraints in order not to disturb the target program execution. Namely, they must avoid using signal management routines, and non-reentrant library routines. Also, the use of dynamically allocated memory [13] (e.g. `malloc`, buffered I/O, ...) will perturb the behavior of the original application.

Annulled instructions The SPARC ISA features annulled instructions in the delay slots of control-transfer instructions (CTIs). These annulled instructions are not emulated and the user analysis routines are never called for them. However, CTIs whose delay slot is annulled are marked as such. The analysis routines can therefore process the CTI and analyze the annulled instruction in the delay slot, within the same call.

Checkpoints The first instruction of a checkpoint is processed like any other program instruction, but it is marked as being such. Consequently, user analysis routines may compute it appropriately. Other checkpoint instructions are not emulated.

5.5.5 Existing Implementations

We implemented user analysis routines to trace instruction and data addresses for "inner", "shm", "ra", and "hc" switching event types, with the support for original instruction address recomputation. The trace gathered is buffered, and output in a binary format to a trace file. We also implemented the user analysis routines which allowed us to collect the statistics shown in Table 1.

In addition, DICE has been modified to simulate an aggressive speculative processor, in order to study the effects of the speculative execution on data caches [18].

[13] The stack is not concerned, here, because of its specific support (see 5.3).

6 Performance Evaluation

Programs instrumented with **calvin2** and linked with DICE have two modes of execution: the fast mode and the emulation mode. In order to evaluate execution slowdowns incurred by both execution modes, we present, in the next two subsections, execution times of the SPEC95 benchmarks, running entirely either in fast mode (6.1), or in emulation mode (6.2).

Then, Subsection 6.3 discusses performance issues to do statistical trace sampling using **calvin2** + DICE, or Shade. In particular, we determine an estimation of the sampling ratio, below which, using **calvin2** + DICE is more interesting than using Shade, in terms of performance.

Note that all the execution times shown were collected as follows: the base execution time is the shortest of three runs; other execution times were selected among three runs by discarding the shortest and the longest ones. Furthermore, execution slowdowns are computed as follows:

$$\text{Slowdown} = \frac{\text{Exec\_time}_{\text{tested}}}{\text{Exec\_time}_{\text{base}}}$$

where each *Exec_time* is the sum of user and system execution times.

6.1 Fast Mode Performance Evaluation

The fast mode execution times presented here were collected running the SPEC95 benchmarks instrumented with either "inner", "ra", "shm", or "hc" switching event types. For all the switching event types but "ra", the checkpoints were inserted inside each procedure call and inside each loop, for a better statistical switching accuracy, as discussed in 4.1. The programs running with the "ra" switching event type were only instrumented with checkpoints inside procedure calls, because there is no need to put any checkpoint inside loops (see 4.2.3).

Because of the "hc" switching event type (see 4.2.4), execution times for the fast mode were collected on a Sun Ultra I workstation, equipped with a 167 MHz UltraSPARC I, with 256 MB of memory, and running a modified Linux operating system (kernel 2.2.10, and RedHat 6.0 distribution). Each benchmark was compiled with `gcc` (version egcs-2.91.66) or `g77` (front-end version 0.5.24) and with the `-O3` optimization option. All the programs were run with the *ref* input data set, except `gcc` for which only one of the 55 executions in the *ref* input data set was performed.

Results shown in Table 2 are execution times of instrumented SPEC95 benchmarks running in fast mode only. The numbers in parentheses represent the slowdowns incurred by the light instrumentation.

The last column of Table 2 reminds the estimated execution slowdowns (computed at the end of Section 4.1, in Table 1) with the associated error ($100 \times \frac{real\_time - Est.}{real\_time}$), with respect to the "inner" and "shm" switching event types execution times.

As expected, the execution overheads are quite low: from 1.07 to 3.09, when the instrumented programs have checkpoints at each procedure call, and inside each loop. Also,

	Base	**"inner"**	**"shm"**	**"ra"**	**"hc"**	**Est. (Errors %)**
compress95	325	541 (1.66)	564 (1.73)	416 (1.28)	978 (3.00)	1.65 (+0.6 / +4.6)
gcc	9	12 (1.36)	13 (1.40)	10 (1.08)	17 (1.92)	1.44 (−5.9 / −2.9)
go	232	331 (1.43)	345 (1.49)	277 (1.19)	487 (2.10)	1.41 (+1.4 / +5.4)
ijpeg	313	355 (1.13)	368 (1.17)	316 (1.01)	464 (1.48)	1.25 (−10.6 / −6.8)
li	500	822 (1.64)	816 (1.63)	609 (1.22)	1301 (2.60)	1.70 (−3.7 / −4.3)
m88ksim	451	762 (1.69)	960 (2.14)	472 (1.05)	1388 (3.09)	1.63 (+3.5 / +23.8)
perl	217	272 (1.25)	258 (1.19)	261 (1.20)	337 (1.55)	1.26 (−0.8 / −5.9)
vortex	598	735 (1.23)	759 (1.27)	691 (1.16)	1006 (1.68)	1.31 (−6.5 / −3.1)
Avg.	**331**	**479 (1.45)**	**510 (1.54)**	**381 (1.15)**	**747 (2.26)**	1.46 (−0.7 / +5.19)
applu	684	895 (1.31)	898 (1.31)	706 (1.03)	1211 (1.77)	1.33 (−1.5 / −1.5)
apsi	495	579 (1.17)	582 (1.18)	511 (1.03)	720 (1.45)	1.28 (−9.4 / −8.5)
fpppp	1495	1636 (1.09)	1708 (1.14)	1502 (1.01)	1592 (1.07)	1.02 (+6.4 / +10.5)
hydro2d	608	796 (1.31)	811 (1.34)	616 (1.02)	1134 (1.87)	1.42 (−8.4 / −6.0)
mgrid	701	806 (1.15)	816 (1.17)	702 (1.00)	990 (1.42)	1.12 (+2.6 / +4.3)
su2cor	395	460 (1.16)	462 (1.17)	401 (1.01)	565 (1.43)	1.22 (−5.2 / −4.3)
swim	430	497 (1.16)	498 (1.16)	456 (1.06)	586 (1.36)	1.17 (−0.9 / −0.9)
tomcatv	460	487 (1.06)	499 (1.09)	463 (1.01)	630 (1.37)	1.17 (−10.4 / −7.3)
turb3d	1893	2270 (1.20)	2272 (1.20)	1907 (1.01)	2921 (1.54)	1.35 (−12.5 / −12.5)
wave5	468	584 (1.25)	588 (1.25)	473 (1.01)	781 (1.67)	1.42 (−13.6 / −13.6)
Avg.	**763**	**901 (1.18)**	**913 (1.20)**	**774 (1.01)**	**1113 (1.46)**	1.25 (−5.9 / −4.17)

Table 2: SPEC95 fast mode execution times (in sec.).

when the checkpoints are only at procedure calls, like with the "ra" switching event type, the execution overheads are even lower (from 1.00 to 1.28), especially running the floating-point benchmarks which make relatively few procedure calls. With the "hc" switching event type, the checkpoint code is more important, so execution slowdowns are globally higher.

The estimated slowdowns given in the last column give a good order of magnitude: only a few values have an absolute error of more than 10 %.

For the integer benchmarks, the estimation is remarkably accurate, except for m88ksim for which it is optimistic[14] (error = +23.8 % with the "shm" switching event type). On an UltraSPARC, the data cache is only 16 KB, direct mapped. We suspect that the relatively inaccurate prediction for m88ksim is related either to a nasty placement of the shared variable in the data cache or a ping-pong phenomenon on the instruction cache.

In contrast, the slowdown estimation is quite pessimistic for a few floating-point benchmarks and ijpeg. Here, CPI performance for the checkpoint code may be higher because 1) the checkpoint code is highly independent from the application code, and 2) added conditional branches are easily predictable because mode switches rarely occur (never in our test).

[14]The same applies to fpppp.

6.2 Emulation Mode Performance Evaluation

In order to evaluate the emulation mode performance, we gathered execution times running instrumented versions of the SPEC95 benchmarks (**calvin2** + DICE), in emulation mode only. Moreover, we compared these execution times with the original programs simulated by Shade analyzers (we further detail each analyzer before we reference it). This allowed us to relate our approach and our tools to a widely spread tool like Shade, and to quantify their differences.

Since Shade can only run with a SunOS operating system, execution times for the emulation mode were collected on a Sun Ultra 60 workstation with two 359 MHz UltraSPARC II processors, and 512 MB of memory, running Solaris 2.7. Each benchmark was compiled with `gcc` (version 2.8.1) or `g77` (version egcs-2.91.66 with front-end version 0.5.24) and with the `-O3` optimization option.

Due to CPU time considerations, we used modified *train* input data sets to obtain a few seconds execution time running the non-modified programs. This was needed to perform executions with full tracing capabilities enabled, in a reasonable amount of time.

6.2.1 Raw Emulation

To estimate DICE raw emulation overhead, we disabled, at compile time, its tracing capabilities (see 5.5.2). Then, we ran instrumented (with **calvin2**, and the "inner" switching event type) programs in emulation mode only, from beginning to the end.

Shade raw emulation execution overhead was measured with an *ad-hoc* analyzer which only loads and simulates the target program without any particular trace control setting.

Table 3 gives the execution times and slowdowns (in parentheses) for the SPEC95 benchmarks. The last column points out the factor DICE is slower than Shade. As DICE does not implement any optimization mechanism, Shade is much faster to make raw emulation. We can notice that Shade buffering mechanisms take a better advantage of floating-point code (CFP95) which is more regular than integer code (CINT95). So, the performance difference between Shade and DICE is higher when running the CFP95 benchmarks (up to 17+ times).

6.2.2 Tracing Instruction Addresses and Data References

Emulation without any simulation makes no sense. For most applications in micro-architecture, the simulation overhead is far higher than the emulation overhead. This seriously lowers the need for emulation optimization. We illustrate this with simply tracing instruction and data references.

For this experiment, DICE was compiled with a level of detail necessary to enable tracing of instruction addresses and data references. The user analysis routines we provided make instruction address recomputation (see 5.5.3), buffer the trace in a 8 K references buffer,

	Base	Shade	DICE	DICE/Shade
compress95	2.0	27.7 (13.85)	198.8 (99.40)	7.18
gcc	4.9	92.9 (18.96)	353.0 (72.04)	3.80
go	1.8	24.7 (13.72)	157.2 (87.33)	6.36
ijpeg	3.1	61.1 (19.71)	298.2 (96.19)	4.88
li	2.7	42.4 (15.70)	219.3 (81.22)	5.17
m88ksim	1.2	20.2 (16.83)	99.5 (82.92)	4.93
perl	1.3	20.1 (15.46)	98.5 (75.77)	4.90
vortex	2.0	35.5 (17.75)	162.0 (81.00)	4.56
Avg.	**2.4**	**40.6 (16.92)**	**198.3 (82.62)**	**4.88**
applu	1.4	11.4 (8.14)	151.1 (107.93)	13.25
apsi	2.0	22.6 (11.30)	154.8 (77.40)	6.85
fpppp	1.3	6.1 (4.69)	106.1 (81.62)	17.39
hydro2d	2.1	18.9 (9.00)	173.3 (82.52)	9.17
mgrid	1.6	16.1 (10.06)	191.2 (119.50)	11.88
su2cor	1.3	13.5 (10.38)	117.8 (90.62)	8.73
swim	1.8	10.5 (5.83)	121.2 (67.33)	11.54
tomcatv	1.7	25.3 (14.88)	138.2 (81.29)	5.46
turb3d	5.3	73.5 (13.87)	380.0 (71.70)	5.17
wave5	2.1	13.4 (6.38)	124.3 (59.19)	9.28
Avg.	**2.1**	**21.1 (10.05)**	**165.8 (78.95)**	**7.86**

Table 3: SPEC95 raw emulation execution times (in sec.).

and store the buffer to a file[15] in a binary format. The Shade analyzer we tested made the same things, except instruction address recomputation[16].

Table 4 represents the execution times for this experiment. The "Shade WT" column presents Shade execution times without any user trace processing: Shade is set up to trace instruction addresses and data references, and the user analysis routine is called but it returns immediately. The overhead measured in this column is the overhead needed to simulate the tested program with tracing enabled, but without actually tracing it, like one would do between trace samples when doing trace sampling (see 6.3 for more details). Also, this overhead can be viewed as a Shade fast mode overhead.

When tracing is enabled and actually performed (columns "Shade" and "DICE" from Table 4), the differences between DICE and Shade shown column "DICE/Shade" narrow, compared to Table 3: the performance gap is now lower than a factor of 2. This shows that DICE is usable as-is, and does not really need aggressive optimizations or complex mechanisms, especially if it is used with time consuming on-the-fly simulators.

[15] We used the file `/dev/null` to remove disk I/O overhead and to lower the operating system activity.

[16] By default, the Shade analyzers we used mapped program code and data at their native locations. However, code and data segment coming from shared libraries seemed to be mapped at unusual addresses.

	Base	Shade WT	Shade	DICE	DICE/Shade
compress95	2.0	33.6 (16.80)	185.7 (92.85)	302.6 (151.30)	1.63
gcc	4.9	136.1 (27.78)	412.1 (84.10)	518.3 (105.78)	1.26
go	1.8	33.5 (18.61)	148.7 (82.61)	223.4 (124.11)	1.50
ijpeg	3.1	68.0 (21.94)	351.5 (113.39)	415.1 (133.90)	1.18
li	2.7	51.2 (18.96)	209.1 (77.44)	323.8 (119.93)	1.55
m88	1.2	21.7 (18.08)	94.4 (78.67)	138.1 (115.08)	1.46
perl	1.3	26.8 (20.62)	103.5 (79.62)	141.3 (108.69)	1.37
vortex	2.0	44.5 (22.25)	166.0 (83.00)	232.9 (116.45)	1.40
Avg.	**2.4**	**51.9 (21.63)**	**208.9 (87.04)**	**286.9 (119.54)**	**1.37**
applu	1.4	17.5 (12.50)	132.6 (94.71)	230.5 (164.64)	1.74
apsi	2.0	29.5 (14.75)	166.7 (83.35)	241.7 (120.85)	1.45
fpppp	1.3	12.0 (9.23)	96.0 (73.85)	169.0 (130.00)	1.76
hydro2d	2.1	24.4 (11.62)	138.7 (66.05)	219.6 (104.57)	1.58
mgrid	1.6	21.5 (13.44)	187.3 (117.06)	305.6 (191.00)	1.63
su2cor	1.3	17.7 (13.62)	107.7 (82.85)	162.8 (125.23)	1.51
swim	1.8	13.6 (7.56)	104.6 (58.11)	167.2 (92.89)	1.60
tomcatv	1.7	31.9 (18.76)	142.2 (83.65)	164.8 (96.94)	1.16
turb3d	5.3	85.2 (16.08)	442.8 (83.55)	571.2 (107.77)	1.29
wave5	2.1	17.5 (8.33)	114.3 (54.43)	194.3 (92.52)	1.70
Avg.	**2.1**	**27.1 (12.90)**	**163.3 (77.76)**	**242.7 (115.57)**	**1.49**

Table 4: Execution times (in sec.) to trace instruction and data references of the SPEC95 benchmarks (emulation mode).

6.3 Trace Sampling

In practice, simulations are not performed on the entire program execution. Very often, they are performed on a big sample taken after trying to skip the initialization phase of the application. Also, trace sampling [14] is used to select samples of execution. In this subsection, trace sampling will designate either statistical trace sampling, either partial execution tracing or simulation. The *sampling ratio* is defined as the amount of simulated activity over the total program activity.

We can roughly estimate the execution slowdown of a trace-sampled program (S_{ts}) as a function of the sampling ratio (r), the execution slowdown of the program running in fast mode ($S_{\text{fast\_mode}}$), and the execution slowdown in emulation mode ($S_{\text{emul\_mode}}$) by:

$$S_{\text{ts}} \simeq rS_{\text{emul\_mode}} + (1-r)S_{\text{fast\_mode}}$$

Note that $S_{\text{emul\_mode}}$ is composed of the emulation slowdown as well as the slowdown needed to export the trace or to make an on-the-fly simulation.

With the data collected in 6.1 and 6.2, we quantify this estimate to compare trace sampling performance of **calvin2** + DICE, and Shade, when tracing instruction and data addresses. Results are displayed in Figure 6 as a function of the sampling ratio:

$$S_{\text{ts}} \simeq S_{\text{fast\_mode}} + (S_{\text{emul\_mode}} - S_{\text{fast\_mode}})r \quad (1)$$

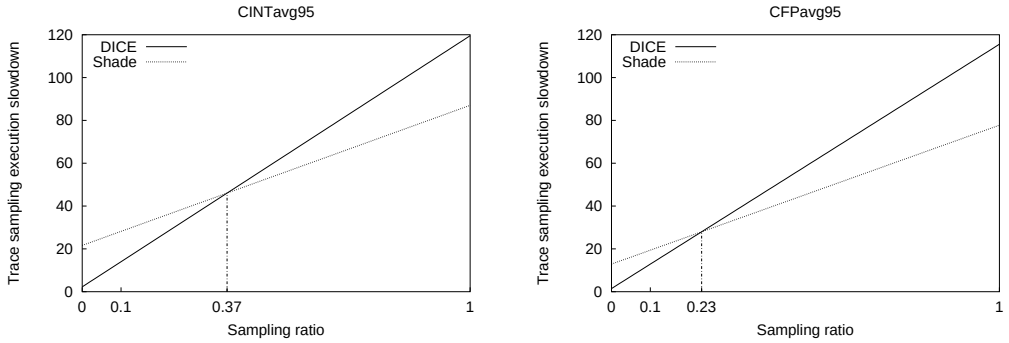

Figure 6: SPEC95 estimated execution slowdowns for trace sampling (instruction and data addresses).

In average, for the CINT95 (resp. CFP95) programs, if the sampling ratio is less than 0.37 (resp. 0.23), then programs running with DICE will be traced faster. Note that these maximum sampling ratio estimates are the same for time consumming simulations, since the simulation time is added to the emulation time (i.e. the simulation slowdown is added to the emulation mode slowdown) whatever tool is used to gather the trace information.

Most studies about trace sampling [8, 9, 12, 14, 17] exhibit good results with sampling ratios of 10 % (or even less) for simulations of generally small caches. So, even if our estimates may be optimistic, **calvin2** + DICE may really be more efficient than Shade, with a sampling ratio of about 10 %.

Moreover, in order to maintain reasonable simulation times, the larger the traced programs, the lower the sampling ratio. To simulate several cache configurations on-the-fly, an additional slowdown of, say, 100 in emulation mode is still optimistic [24]. Given a one hour general purpose workload (c.f. CINT95), and a low sampling ratio of, say, 5 %, using equation (1), and data from Table 4, the cache simulation with Shade would require about 35 hours. Using **calvin2** + DICE ("hc" switching event type), the same simulation would require about 22 hours. Similarly, to simulate a complex microprocessor (with an optimistic additional slowdown of, say 1000 [3, 11]) on the same program with a sampling ratio of 1 %, Shade would take about 33 hours, while **calvin2** + DICE would take about 15 hours.

7 Summary and Future Work

In this paper, we have presented a new approach for running on-the-fly architecture simulations. Our approach combines light static code annotation and instruction-set emulation.

The light static code annotation inserts some *checkpoints* in the code and provides target programs with a fast (direct) execution mode. This mode is used to rapidly position the execution in interesting tracing states. *Switching events* make the inserted checkpoints switch the execution to an emulation mode. The emulation mode is managed by an embedded instruction-set emulator which makes tracing/simulation possible. We implemented a static code annotation tool, called **calvin2** to make the fast mode, and a SPARC V9 instruction-set emulator called DICE to manage the emulation mode. We evaluated the performance of both tools and compared it with the state of the art in dynamic translation, namely Shade.

The fast mode execution slowdowns for the SPEC95 benchmarks ranged from 1.01 to 3.09, depending on the switching event type used. Such execution slowdowns make it possible to skip large portions of long running workloads before entering the emulation mode to begin the simulation.

In emulation mode, DICE performance, compared to Shade, appeared to be quite poor for raw emulation: DICE has run the SPEC95 benchmarks from 3.8 to 17.4 times slower than Shade. However, our approach deemphasize the need for very efficient raw emulation by providing a very efficient fast mode. Moreover, the relative performance of Shade seriously decreases when both emulators have to generate some real trace: DICE has been from 1.16 to 1.76 times slower than Shade to collect instruction and data addresses. This result shows that, for a realistic use, DICE does not really need aggressive optimizations.

Using the execution slowdowns collected in fast mode and in emulation mode, we estimated the execution slowdowns incurred by **calvin2** + DICE, and Shade when tracing/simulating partial program activity (trace sampling). This estimate showed that if the sampling ratio is less than 37 % (resp. 23 %), then, on average, the CINT95 (resp. CFP95) programs are more rapidly traced by **calvin2** + DICE than by Shade. These results greatly justify our approach, since, even if these estimates may be optimistic, lower sampling ratios (about 10 %) are used to do statistical trace sampling [8, 9, 12, 14, 17], and even lower sampling ratios may be used in practice to simulate long running applications.

The tools presented in this paper can only trace user-level activity, not the operating system. Using such uncompleted workloads may lead to misleading conclusions [1, 25, 26]. For this reason, DICE has been extended to be embedded in a Linux kernel operating system. This extension of DICE is called LiKE (Linux Kernel Emulator) and can emulate partial operating system activity, namely the core of the system calls. In a near future, we plan to work on LiKE to make it possible to simulate most of the operating system activity.

Also, enabling complete execution-driven simulations with DICE is one of our main concerns. This will allow us to simulate realistic microprocessors (e.g. speculative, out-of-order execution, ...) on long running applications.

Finally, the "ra" switching event type is very efficient to trace given routines of the target program. So we will also study how **calvin2** + DICE could be used for program optimizations.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. Cache performance of operating systems and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[3] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] D. C. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-96-1308, University of Wisconsin, Madison, July 1996.

[5] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proceedings of the 26nd Annual International Symposium on Computer Architecture*, pages 64–74, May 1999.

[6] M. J. Charney and T. R. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 41(3), 1997. `http://www.almaden.ibm.com/journal/rd/413/charney.html`.

[7] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[8] P. J. Crowley and J. L. Baer. Trace sampling for desktop applications on windows NT. In *Proceedings of the Micro Workshop on Workload Characterization*, November 1998.

[9] J. W. C. Fu and J. H. Patel. Trace driven simulation using sampled traces. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 27th Hawaii International Conference on System Sciences. Volume 1 : Architecture*, pages 211–220, Los Alamitos, CA, USA, January 1994. IEEE Computer Society Press.

[10] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[11] Stephen Alan Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.

[12] Richard E. Kessler, Mark D. Hill, and David A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transaction on Computers*, 43(6):1325–1335, June 1994.

[13] T. Lafage, A. Seznec, E. Rohou, and F. Bodin. Code cloning tracing: A "pay per trace" approach. In *Euro-Par'99*, Toulouse, August 1999.

[14] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, 1988.

[15] Alvin R. Lebeck and David A. Wood. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):42–77, January 1997.

[16] M. H. Lipasti and J.P. Shen. Exceeding the dataflow limit with value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[17] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Effectiveness of trace sampling for performance debugging tools. In Blaine D. Gaither, editor, *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, volume 21-1 of *Performance Evaluation Review*, pages 248–259, New York, NY, USA, May 1993. ACM Press.

[18] S. Mohamed. Simulation of a processor with speculative execution. DEA report, June 1999.

[19] T. Puzak. *Analysis of cache replacement algorithms.* PhD thesis, University of Massachusetts, 1985.

[20] E. Rohou, F. Bodin, and A. Seznec. SALTO: System for assembly-language transformation and optimization. In *Proceedings of the Sixth Workshop Compilers for Parallel Computers*, December 1996.

[21] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[22] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[23] S. W. Sathaye. Mime: A tool for random emulation and feedback trace collection. Master's thesis, University of South California, 1994.

[24] R. Uhlig and T. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 1997.

[25] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Trap-driven simulation with Tapeworm II. In *Proceedings of the ASPLOS-VI Architectural Support for Programming Languages and Operating Systems*, pages 132–144, San Jose, California, October 1994.

[26] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[27] *UltraSPARC User's Manual.* Sun Microsystems, july 1997. `http://www.sun.com/microelectronics/manuals/ultrasparc/802-7220-02.pdf`.

[28] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, Version 9.* SPARC International, Inc., 1994.

[29] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, pages 68–79, New York, May 1996. ACM Press.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399