SIND: Sind Is Not Dynamo

Dino Dai Zovi <ghandi@cs.unm.edu>
Trek Palmer <tpalmer@cs.unm.edu>

May 9, 2001

1 Introduction

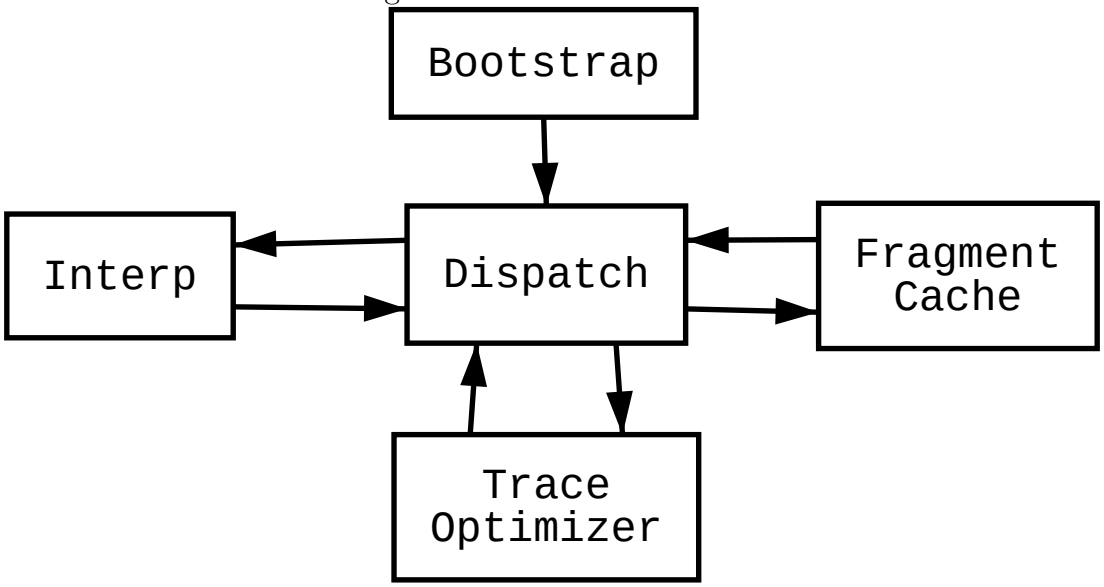
As programs grow in both complexity and modularity, the difficulty of static optimizations increases as the benefits of such static analysis decrease. However, as programs are run their paths of execution lend themselves to optimization at runtime. SIND seeks to dynamically optimize running programs by isolating important segments of code and optimizing them in their runtime context without having to modify the statically optimized program text.

2 Design

SIND was inspired by the Dynamo project at HP and is based on the same ideas. SIND essentially attaches itself to a running process and transparently profiles its execution to identify critical or "hot" segments of the code (a trace). These hot segments are then optimized by a linear pass optimizer which exploits optimizations available only at runtime (see [6] section 8.1). The optimized trace, or fragment, is then placed inside the fragment cache which holds all the fragments for the currently running process. Control is returned to the transparent profiler which will cause execution of the fragment when that point in the code is encountered again. The theory is that, after enough time, the majority of the executed program text will have been converted into fragments, and so the execution will occur mainly in the fragment cache. Execution within the cache will be faster than the execution of the statically optimized binary and will eventually offset the initial cost of the profiler and optimizer.

SIND is a transparent optimization package and so has to fullfill several design goals. First, it must seamlessly and transparently interrupt the execution of a loaded binary, and then hand over control to the transparent profiling and optimizing modules. The transparent profiler must monitor the executing binary without modifying the binary text segment in memory. SIND uses an interpreter for this task. This interpreter must also be able to reliably identify "hot" code segments for optimization and then hand the identified "hot" trace off to the optimizer. SIND accomplished trace identification using a slightly modified form of the speculative trace identification technique described in [5] [6]. Once a trace is created the SIND interpreter invokes the Dispatcher object which handles the communication between

Figure 1: SIND Architecture



the profiling interpreter, the dynamic optimizer (see 2), and fragment cache. The dispatcher takes the trace and hands it to the optimizer which will convert it to a faster running code segment called a "fragment". This fragment is then handed to the Dispatcher for insertion into the fragment cache. The fragment cache holds all the fragments and is called by the Dispatcher to add, remove, and execute fragments. The Dispatcher forms a level of indirection within SIND that guarantees a high level of modularity for the subsystems.

3 Bootstrapper

The system is initially entered through a bootstrapping program. The bootstrapper is given the path to the target executable and any arguments to pass along to it. The primary function of the bootstrapper is to launch the inferior child process, letting it run to a certain point, and then begin running the target in the machine code interpreter. In the implementation of this module, several approaches were investigated using the `ptrace` and `procfs` facilities to control the child process as a breakpoint debugger would.

3.1 Initial Break Point

There were three obvious points in the child process' execution where it would make sense to cut over to the interpereter: just after the `exec` system call returns (as done by the `ptrace` facility), `_start`, and `main`. Several factors influence the decision. For example, after `exec` the execution of userspace code begins in the dynamic linker. Because the dynamic resolution of references and shared library loading happens only once, it would not benefit from the "hot" fragment detection in the interpreter. The same argument could be used for the actions

in `libcrt0` (text beginning at `start`). Therefore, it makes the most sense to switch over to interpretation upon entering the `main` function.

To accomplish this, the address of the `main` symbol is located in located in the ELF executable image. On the Solaris implementation this is done through the `nlist(3E)` API. The instruction is replaced with one that executes a "trace trap" that will be caught by the parent process. This effectively passes control to the parent process (the bootstrapper) that will resume the child process under interpretation.

3.2 `ptrace`

Initially, the Berkeley `ptrace` facility for breakpoint debuggers was used. Of the facilities investigated, `ptrace` is the most portable. However, the system has serious performance limitations. All memory accesses must be done one word at a time through a kernel system call. In addition, in [1], the kernel implementation of `ptrace` is criticized for its poor performance. Instead, the faster and more flexible `procfs` system was used.

3.3 `procfs`

`procfs` is a virtual filesystem that provides a window into the kernel structures controlling a process as well as the process' address space. The various structures and the address space are read from virtual files in the `/proc/`*pid* directories.

The facility provided analogues to the commands available via `ptrace` (under Solaris, `ptrace` is implemented in terms of `procfs` in `libc`). These were used to set the signals, faults, and system calls that stopped the child process and passed control to the parent. The address space virtual file was used to write the breakpoint instruction into the child's text segment and proxy memory read and write requests from the interpreter.

4 Interpretation and Optimization

SIND is based on the concept of transparent optimization, and so must transparently instrument the running binary. SIND uses an interpreter to monitor the executing code without modifying the loaded binary. SIND incorporates an extended version of the ISEM SPARC emulator for this purpose. Upon initial invocation by the dispatcher the interpreter is handed a memory which contains the binary image. The interpreter executes the instructions contained within this memory, profiling very specific types of instructions. The interpreter makes special note of branches and considers backwards taken branches to be potential starting points for hot traces. This stems from the fact that many backwards taken branches are in fact looping statements. The interpreter also flags instructions targeted by fragment exit points as potential hot trace starts.

When the interpreter has identified a hot trace it records the instructions into a buffer until an end-of-trace condition is met. An end of trace is signalled by either: a backwards taken branch or if the trace length exceeds some preset threshold. The interpreter just copies non-branch instructions straight into the buffer, and records target information for branches as well as the instruction itself. This information is later used by the optimizer to eliminate

branches. When the trace collection is finished, the trace is passed to the dispatcher which then relays it to the optimizer.

The optimizer will generate a fragment from the trace by performing several linear time optimizations. This proto-fragment is returned to the dispatcher which hands it off to the fragment cache manager which adds prologue and epilogue segments to the fragment (for linking purposes).

Currently, this is implemented by inheriting from the ISEM base classes. Both the integer unit and the MMU have been extended to accomadate the profiling additions. Currently only the interpreter and the Dispatcher have been implemented.

5 Conclusion

Runtime optimization offers an interesting solution to the problem of increased static optimization overhead. Unfortunatly, the SIND code is still in development and we were unable to run any meaningful experiments. However, several other projects (notably Dynamo) have demonstrated the efficacy of runtime optimization, and we are encouraged by this.

6 Related Work

We, of course, owe a great debt to the HP Dynamo project. We also examined several other dynamic optimization projects, however we found none that were fully free (i.e. GPL). Also the Shade project at Sun Microsystems involved the transparent profiling of SPARC binaries, and offered some insight to the modification of the ISEM interpreter.

7 References

References

[1] Marshall Kirk McKusick et al. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[2] Eric Feigin. *A Case for Automatic Run-Time Code Optimization*. PhD thesis.

[3] David Keppel Robert F. Cmelik. Shade: A fast instruction set simulator for execution profiling.

[4] Smith Traub, Schechter. Ephemeral instrumentation for lightweight program profiling.

[5] Sanjeev Banerjia Vasanth Bala, Evelyn Duesterwald. Dynamo: A transparent dynamic optimization system.

[6] Sanjeev Banerjia Vasanth Bala, Evelyn Duesterwald. Transparent dynamic optimization: The design and implementation of dynamo.