Threaded Software Dynamic Translation Master's Project

Daniel Williams

August 30, 2005

Abstract

Software dynamic translation and multiprocessing hardware are both becoming prevalent in today's computing environment. Unfortunately, many of the programs being run under software dynamic translation are single threaded, and are unable to take advantage of thread level parallelism. This work presents the groundwork for enabling the Strata software dynamic translation framework to run in multiple threads. This allows the system to offload the work of translation into its own thread. Threaded Strata was tested by allowing the builder to speculatively build fragments that were not yet needed by the application. Using this technique threaded Strata able to speculatively build on average 69% of the fragments of the SPEC2000 CINT benchmark suite. Performance on these benchmarks gives us reason to believe that threaded software dynamic translation can be effective method of offloading work of the software dynamic translator.

1 Motivation

Software Dynamic Translation (SDT) is an increasingly popular method of program execution. SDT techniques can be used to implement dynamic optimizers, debuggers, dynamic binary translators, among other systems [1, 2, 7, 3, 5]. At the same time, computer hardware has been moving toward more parallel methods of execution [4]. Symmeteric multiprocessors (SMP) have become commonplace, and multiple core CPUs and simultaneous multithreading (or hyperthreading) are gaining popularity. These hardware architectures are designed to exploit parallelism in multiuser systems, and multithreaded code. However, much of the code in use today is still single threaded, and therefore is mostly unable to take advantage of these advances at the application level. When these single threaded applications are running under a software dynamic translator, the application incurs the overhead of translation while waiting for translation to occur. By enabling our SDT framework to run in a separate thread, we can reduce the overhead of translation on parallel architectures by separating the work of the translator into a thread separate from the thread executing the application. This enables the translator to continue working while the application runs in the other thread.

This paper presents the groundwork for taking advantage of parallelism within a SDT framework by changing the Strata framework to operate cohesively while running in multiple threads. These changes enable Strata's builder to operating in one thread, while another thread is responsible for executing translated application code. The feasibility of threaded translation is tested by speculatively building fragments, that is, building fragments before they are requested by the application. The rest of this paper is organized as follows. Section 2 gives a high level overview of the design of the Strata framework. Section 3 details the design of the code added to run Strata in two threads. Finally section 4 gives experimental results, analysis, and future work.

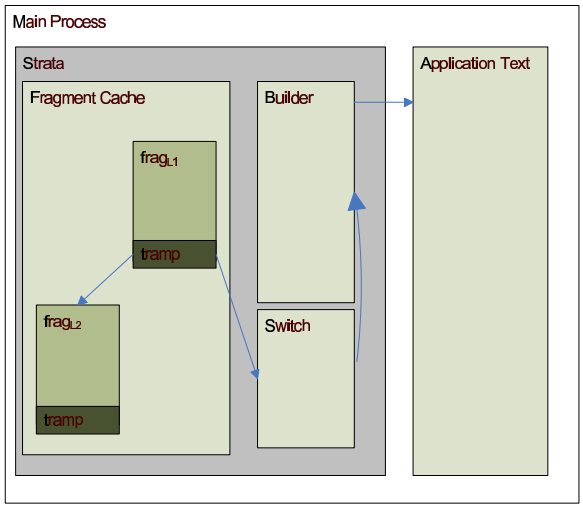
2 Strata Design Overview

Strata is a framework for creating software dynamic translation systems [7]. The basic structure of

Strata is shown in Figure 1. Strata acts as a translation layer between the application and the operating system. It translates the application text, and executes the translated code in the fragment cache. At the end of each fragment, Strata places a trampoline to return control of the application to Strata. Since control can return to Strata at any point in the application's execution, Strata must perform a context switch to save the application state before returning to Strata. Once the application has returned, Strata builds the next fragment needed by the application, restores the application's context, and allows the application to continue executing.

Executing a context switch at every control flow transfer introduces significant overhead to the application. A broad range of overhead reduction techniques is discussed in detail by Scott, et al [6]. The most important of these techniques are fragment linking and indirect branch translation caching (IBTC). When running Strata without fragment linking, every conditional branch returns to Strata to look up its target. If the target has already been built, Strata then does another context switch back to the application. With fragment linking, before it context switches back to the application, it patches the previous fragment so that the next time it is executed, it jumps directly to the target fragment. IBTCs attempt to do similar linking with indirect branches. However, since the branch target is not known at translation time, Strata emits a lookup into a small cache for the target fragment. If the target is not found, it returns control to Strata, and the target is added to the IBTC.

Figure 1: Strata running in a single process



3 Design of Strata with Threads

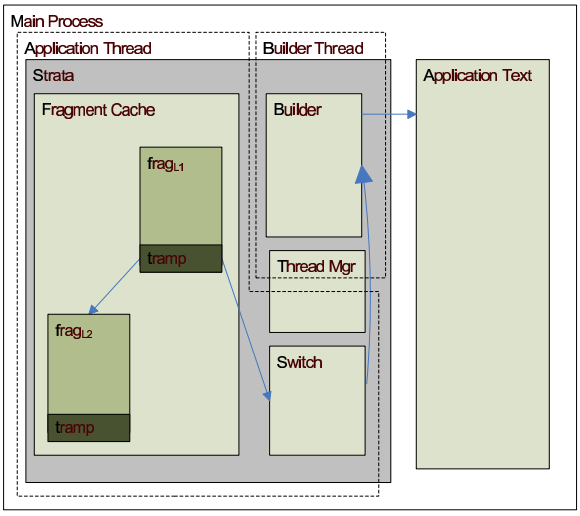
In order to exploit parallelism of multiprocessor systems, the work of translation must be separated from the execution of the application. By decoupling the process of building fragments from the execution of the application, there is a clear separation of duties for both threads. The first thread, referred to here as the Application Thread, is responsible for executing the translated application code. The second thread, the Builder Thread, is responsible for building new fragments. The goal is to allow the Application Thread to behave as much as possible like the orginial application, and encapsulate all the work of translation within the builder thread.

The changes to enable a threaded version of Strata are encapsulated in a new module called the Thread Manager. The Thread Manager uses the POSIX Thread (pThreads) interface to implement threading in a machine independent way. It is responsible for maintaining coherence of the fragments. Specifically, it ensures that the application thread only enters fully constructed fragments, and that application thread is not waiting for a fragment while the builder builds a different fragment. The Thread Manager is also responsible for ensuring that the builder doesn't overuse resources.

3.1 Thread Manager

The Thread Manager is the interface between to the Application Thread and the Builder Thread, and it controls communication between the threads. As shown in Figure 2, the Thread Manager runs in both the Builder and Application thread. This section describes the details of the Thread Manager's implementation, including inter-thread communication, thread locking, and speculation.

Figure 2: Strata running in separate threads



In a single threaded environment, there are three operations that the builder is must complete before returning to the application. First it must determine what, if any, fragments to build. Second, it must build those fragments, and finally, it must return control to the location of the fragment to be executed next. When using a separate Builder Thread, these operations are split between the Builder and the Application threads. The Application Thread first determines if a fragment must be built by looking up the target address in the fragment cache. If the target fragment does not exist the Application Thread enqueues the address to the builder, then waits until the required fragment is built. Once the fragment is completed the Application thread is woken up and it performs a context switch back to the application.

The Builder Thread runs a loop described by the pseudo code in Figure 3. First, the builder attempts to do speculation, if possible. Speculation is described in detail in Section 3.3. If speculation is not possible it goes into a rest state. After it builds a speculative fragment or is woken from its rest state, it checks the builder queue to see if the application is waiting on a fragment to be built. This design allows many fragments to be built speculatively, checking after each fragment that the application has not requested that a different fragment

Figure 3: Pseudo code of the Builder Threads main loop

```
while True
    if speculation is possible
        build_fragment()
    else
        builder_rest()
    if the application is waiting
        build_fragment()
```

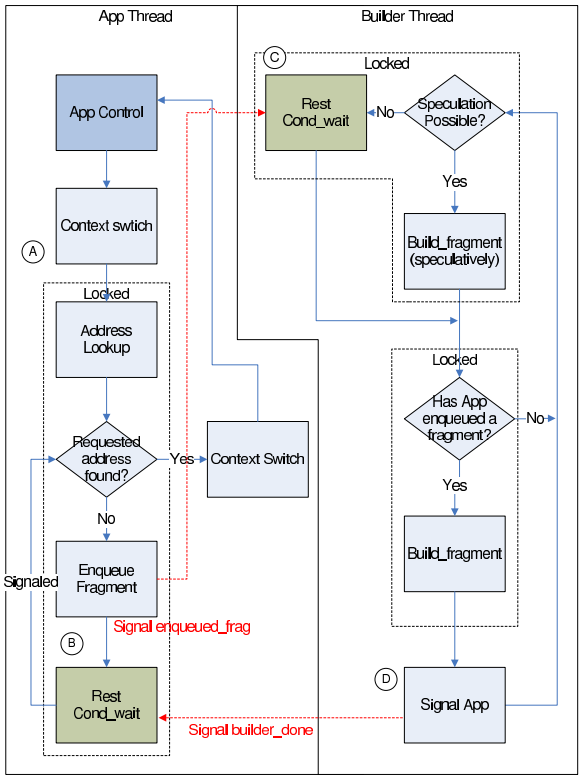
be built. Once the application request a fragment, that fragment is always built next, minimizing the time the application spends waiting for a fragment.

3.2 Thread Locking

Locking between the Application and Builder threads is carefully designed to allow as much parallelism as possible without corrupting any data. The Thread Manager uses two condition variables, and an associated mutex to implement locking. The condition variables are `enqueued_frag` and `builder_done`. Figure 4 is a diagram showing the locking of the relevant critical sections.

When the Application thread returns to Strata, symbol (A) in Figure 4, it first obtains the mutex, then enqueues the address of the next fragment to execute, which signals the `enqueued_frag` condition. At (B) the Application Thread enters a `pthread_cond_wait()` for the `builder_done` variable. The sematics of the `pthread_cond_wait()` are to release the lock once the thread goes into a wait state, and to reacquire the lock, after the condition has been signaled. When the `builder_done` condition is signaled by the builder (D), the Application Thread looks up the requested fragment again, to insure that is has been built. This is to ensure that the Application was woken up for the correct fragment. If it has not been built, the Application Thread signals the Builder Thread again to indicate that it is still waiting on a fragment, then returns to a `pthread_cond_wait()`. When the Builder Thread recieved the `enqueued_frag` condition at (C), it begins building the fragment enqueued by the Application Thread. Once the Builder

Figure 4: Control Flow Graph of the Thread Manager, showing thread locking



Thread has finished building, it either enters `pthread_cond_wait()` waiting on `enqueued_frag`, or it begins building speculative fragments. Speculation is described more in the next section.

3.3 Thread Manager Speculation

By creating a separate thread that simply waits for a signal from the Application Thread, builds that fragment, and returns to a conditional wait, Strata performs each build in lock-step. This causes the threaded version of Strata to essentially behaves the same as the single threaded version. However, if the builder is running in a separate thread, it can use otherwise idle processing time to do useful work. This paper explores using speculation, specifically conditional branch speculation and call-site speculation. Conditional branch speculation attempts to build targets of conditional branches before they are requested by the application. Call site speculation attempts to build fragments corresponding to the return address of a call.

3.3.1 Speculation Design

The goal of speculatively building fragments is to build as many useful fragments as possible without interrupting the running of the application. To achieve this, potentially useful fragments are added to a work list as they are discovered. The work list is a priority queue of unbuilt fragments. When an unbuilt fragment is added to the work list it is given the highest priority, and placed after all other fragments with equal priority. Each time the application requests a fragment, all the fragments that are still in the work list have their priority lowered. When a fragment is built speculatively, the addresses of other potential fragments might be found. Sections 3.3.2 and 3.3.3 describe how these addresses are found. Once these addresses are found, they are added to the queue at the highest priority, after all other fragments at that priority. This means that fragments closer to the fragment last requested by the application will be built first. Speculation continues until either the application returns requesting an address that has not been built, or until the work list becomes empty.

There were a number of modifications needed to efficiently support this type of speculation. The work lists are built using the preexisting struct `strata_fragment_list`. This struct was modified slightly to add a priority value. Also, fragments had to be changed to allow for the creation of "unbuilt" fragments. When Strata runs in a single thread, there is no need to have an unbuilt fragment, since once a fragment was started, there could be no interruptions before it finished. In an environment with multiple threads, it is possible to lookup fragments that are not completely finished. Thus there must be some way to delineate built and unbuilt fragments. This is achieved by separating the fragment creation process into two functions, `strata_create_fragment()` and `strata_begin_fragment()`. `strata_create_fragment()` builds a fragment stub, with a target application PC, but no other data. Then when the fragment is actually built,

`strata_begin_fragment()` is called, initializing the rest of the fragment with the correct values, including a beginning fragment PC. In order to insure that no stub fragments are used by the application, a new flag was added, `STRATA_FRAG_READY`, which is only set once fragment has been built, and is ready to be used by the application.

3.3.2 Conditional Branch Speculation

When a fragment is being built, there are a number of instructions that can cause Strata to terminate the fragment. The most common is conditional branches. Conditional branch speculation attempts to reduce the number of context switches into Strata by linking in the targets of conditional branches that haven't been requested by the application yet. If the fragment ends in a conditional branch, Strata adds both the branch target and the fall through address to a work list. Since unconditional branches are inlined in the current version of Strata, this covers all control flow except for indirect branches. Call site speculation attempts to address one part of indirect control flow.

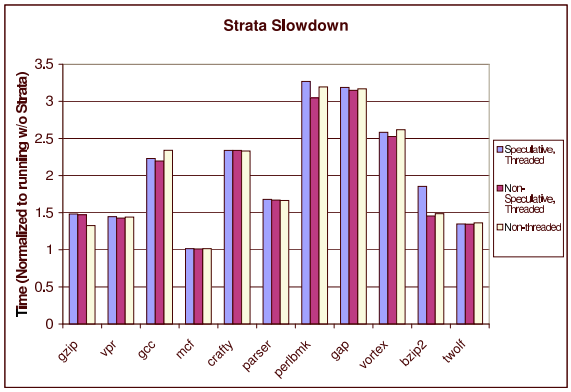
3.3.3 Call site speculation

When Strata translates a call, it performs partial inlining, where the return address is manually pushed onto the stack, and Strata continues translation of that fragment at the address of the called function. When the return statement is reached, it is treated as an indirect branch, and the application will look up the return address in the Indirect Branch Translation Cache, and if it finds a hit, it will jump directly to the fragment corresponding to the return address. In a well behaved program, the application will return to the statement after the call. This presents another place to perform speculation. When Strata translates a call, it adds the return address to its work list.

3.4 Resource Management

In order for the application to perform as well as possible, the Builder Thread must not consume CPU resources that would otherwise be used by the Application Thread. The Thread Manager achieves this by only building one speculative fragment at a time, checking the build queue each time to make sure that the application has not returned to Strata. It would be possible to do even more finely tuned handling of resources to insure that the Builder's speculation is unobtrusive. These include using the `yield` system call on the Builder Thread between fragment builds and also lowering the priority of the Builder Thread.

Figure 5: Strata Slowdown



4 Results

Moving the work of fragment building to a separate thread is intended to allow the application to run faster because it does not have to spend time waiting for each fragment to be built. However, because fragment building is a minor source of building overhead, the total execution time is not reduced. Figure 5 shows the SPEC2000 CINT timings on a hyperthreaded Pentium 4 normalized to the time taken to run the benchmark without Strata. This figure compares timings for running Strata in a single thread. Except for gcc, adding a second thread without doing any fragment speculation slowed execution down, though only very slightly. In general, when speculative fragments were built, performance also decreased slightly. There are a number of possible explanations for this. Thread scheduling might have scheduled the Builder Thread at the expense of the Application Thread. It is also possible that the addition of unused speculative fragments degraded

Figure 6: Percentage of fragments created speculatively.

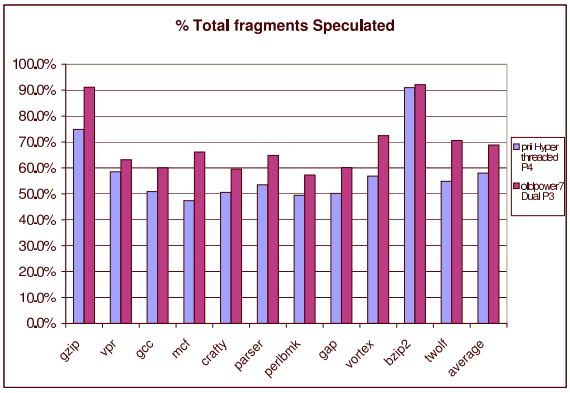
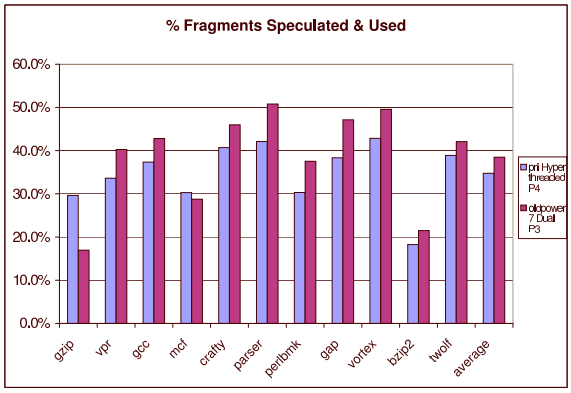
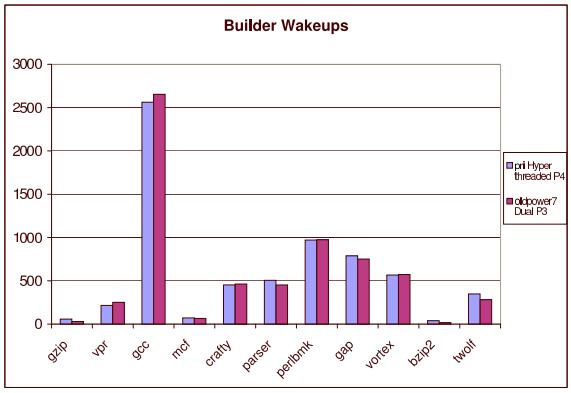


Figure 7: Percentage of fragments created speculatively and also used.



I-cache performance.

Figure 8: Number of builder wakeups



4.1 Speculation Performance

To show the feasibility of running Strata in a separate thread, the system was configured to build as many fragments speculatively as possible. Figure 6 show the percentages of fragment that were built speculatively. On average, Strata was able to build 58% of its fragments speculatively on the hyperthreaded machine, and 69% on the SMP machine. With SMP, Strata was able to build over 90% of the fragments for gzip and bzip2.

One drawback of speculative building of fragments is that not all fragments that are built are necessarily used. Figure 7 shows how often the speculative fragments were used. Specifically, it shows the percentage of fragments that were built speculatively, then later used by the application. On average, just over a third of the running program was built speculatively. Also gzip and bzip2, which had very high percentages of fragments speculated, both had a relatively low percentage of fragments both speculated and used, 17% and 22% respectively. This is due to gzip and bzip's relatively small working set. The builder was able to speculate about most of the program, but only a small percentage of it was actually executed. This indicates that there is room for improvement with regard to how the builder spends its time when its not building fragments requested by the application.

When Strata attempts to speculatively build fragments, it continues processing fragments on its work list until the work list is empty. If the builder creates all the fragments on its worklist and the application has not yet returned to Strata, the Builder Thread goes to sleep. Once the application returns it signals the builder, which wakes up the Builder Thread. Figure 8 shows the number of times a builder went to sleep on a condition wait. On the hyperthreaded system the builder went to sleep an average of 598 times, and on the SMP system it went to sleep an average of 592 times.

4.2 Analysis

Although the timing results show no significant performance increases, the other data collected gives reason to believe this architecture can be beneficial way to implement a software dynamic translator. Building fragments speculatively is an important first step, showing that code can be created and used without the direct intervention of the Application thread. The builder is able to build a significant number of the fragments speculatively, and also went into a waiting state a number of times. These statistics indicate that the speculative building was not underperforming because of lack of time, but rather, lack of effective work to do. Therefore, if the builder spent its time doing more work to improve the application's translated code, performance improvements could be seen. Section 4.3 describes other possible work that could be done by the builder that may be more beneficial.

4.3 Future Work

There are many potential advantages to running the builder in a separate thread that could give better performance results than simply building speculatively. It has been observed that most of the overhead incurred by an application running under software dynamic translation results from indirect branches for fragments that have already been built, but were not accurately predicted by the dynamic translator. It would be possible for the builder thread to use it resources to populate the IBTC to reduce the context switches that result from indirect branch targets that are already built.

Another area of fruitful future work is using builder to do more advanced optimizations. Optimizations done by software dynamic translators are generally kept very lightweight so the time taken to do the optimization doesn't overshadow the performance gains of the optimization. By performing the optimization in a separate thread, it would enable the SDT to perform optimizations that require larger amounts of analysis because this analysis can occur in parallel with execution of the application. For example, Pin [5] does register re-allocation to harvest registers for use by the dynamic translator. The analysis to accurately find unused registers is relatively expensive, but it would be possible to create fragments that simply spills needed registers (as Strata does now), but then allows the builder to analyze and recreate with register re-allocation as the application executes in the other thread.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000.

[2] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, 2003.

[3] K. Ebcioǧlu and E. R. Altman. Daisy: dynamic compilation for 100 In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, 1997.

[4] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[6] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. Overhead reduction techniques for software dynamic translation. In *IPDPS Next Generation Software Program - NSFNGS - PI Workshop*, 2004.

[7] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, 2003.