# Techniques and Tools for Dynamic Optimization

Jason D. Hiser†, Naveen Kumar‡, Min Zhao‡, Shukang Zhou†,
Bruce R. Childers‡, Jack W. Davidson†, Mary Lou Soffa†
‡Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260
†Department of Computer Science, University of Virginia, Charlottesville, VA 22904
{hiser,jwd,soffa,zhou}@cs.virginia.edu {childers,naveen,lilyzhao}@cs.pitt.edu

## Abstract

*Traditional code optimizers have produced significant performance improvements over the past forty years. While promising avenues of research still exist, traditional static and profiling techniques have reached the point of diminishing returns. The main problem is that these approaches have only a limited view of the program and have difficulty taking advantage of the actual run-time behavior of a program. We are addressing this problem through the development of a dynamic optimization system suited for aggressive optimization—using the full power of the most beneficial optimizations. We have designed our optimizer to operate using a software dynamic translation (SDT) execution system. Difficult challenges in this research include reducing SDT overhead and determining what optimizations to apply and where in the code to apply them. Another challenge is having the necessary tools to ensure the reliability of software that is dynamically optimized. In this paper, we describe our efforts in reducing overhead in SDT and efficient techniques for instrumenting the application code. We also describe our approach to determine what and where an optimization should be applied. We discuss other fundamental issues in developing a dynamic optimizer and finally present a basic debugger for SDT systems.*

## 1. Introduction

Over the past several decades, the optimizing compiler research community has developed sophisticated, powerful algorithms for a variety of code improvements: register allocation, code motion and partial redundancy elimination, procedure inlining, loop optimizations, memory hierarchy optimizations, and code scheduling to name a few [1].

While there are still promising avenues of research for particular optimizations, research on new or improved optimizations is now at the point of diminishing returns. Execution-time performance gains by a new or improved optimization is unusually small—an improvement of a few percent is typical. With the diminishing returns of current optimization research, the challenge for compiler construction and optimization research is to develop new techniques that yield performance improvements that were typical of early optimization research. Fortunately, recent developments in optimization research indicate that significant performance improvements are possible by taking a radically new approach to the design and construction of compilers.

The key development is that optimizing a program while it executes is both possible and can yield substantial performance improvements. Adapting and applying optimizations as the program executes has several advantages over only applying optimizations statically. The behavior of a running program can change over its lifetime, and thus different optimizations are beneficial at different points during the program's execution. For example, program paths that are frequently executed at certain points in a program's execution may execute infrequently, if at all, at other points of the program's execution. As a concrete example, consider loop unrolling. Knowing a loop's trip count during the various phases of a program's execution can enable more effective application of loop unrolling. During the program phase where the loop trip count is high, the loop should be unrolled as the benefit will be high. The cost in terms of code space is justified. During the program phase where the loop trip count is low or zero, the loop should not be unrolled as the benefit is low and potentially negative if unrolling causes some frequently executed code to be evicted from the instruction cache.

Such behavior is difficult to recognize and capitalize on using offline profile-guided optimization techniques that collects aggregated data from the program's execution using a training set of representative data. The instantaneous behavior of a program running on real data may be very different from the behavior inferred by processing a training data set. In contrast, dynamic optimization can easily adjust to changing program behavior by applying the most appropriate optimizations for a given phase of a program. Furthermore, having accurate information at run time not only determines what optimizations to apply and when, but it also enables more effective application of certain beneficial optimizations. For example, knowing which branch of an if-then-else statement is executed more frequently at different points in a program execution can enable more effective code placement.

1-4244-0054-6/06/$20.00 

While applying optimizations on executing code offers many benefits, to achieve the improvements in performance we desire requires rethinking how we build optimizers. Currently, most optimizers have a rigid structure that is fixed when it is constructed. In currently available optimizers, for example, the suite of optimizations that can be applied as well as the order that they are applied is fixed when the optimizer is built. The ordering is chosen to achieve the best performance over a spectrum of programs (e.g., SPEC2000, MediaBench, etc.). While most optimizers support enabling and disabling optimizations via command line options for the entire application, few optimizers offer any flexibility beyond this. Furthermore, even when flexibility is available, it is up to the software developer to determine which optimizations to apply.

To explore the spectrum of static to dynamic optimizations, we have begun to design, implement, and evaluate an innovative optimization paradigm and algorithms that we predict can achieve significant improvements in run time performance. Our approach is to create a flexible, adaptable optimization system where compile-time and run-time plans are generated automatically by the optimizer using information about the application, the target machine, and cost/benefit of the suite of available optimizations. Our dynamic optimizer is built on a SDT, called Strata. This paper briefly discusses our efforts in developing an aggressive dynamic optimizer. In particular, our contributions discussed in this paper include:

- techniques to reduce the overhead of an SDT system,
- efficient techniques to insert/remove instrumentation,
- optimization techniques for instrumentation code,
- a framework that includes models for code, optimizations and resources useful for predicting the benefits of optimizations without applying them,
- a debugger for an SDT system, and
- identification of special issues in developing dynamic optimizers.

In Section 2, we discuss techniques for reducing the overhead of SDTs and address the cost of instrumentation needed for dynamic instrumentation. In Section 3, we present our framework for predicting the benefit of an optimization and discuss in developing a dynamic optimizer. Section 4 presents our scheme for debugging programs running under our dynamic optimizer. Finally, 5 summarizes our current findings.

## 2. Low Overhead Dynamic Translation

In order for an advanced execution system to monitor and transform an application efficiently, the system must introduce minimal overhead for processor time, memory usage, disk usage, etc. To this end, we developed and evaluated overhead reduction techniques within the Strata software dynamic translation system [2][3].

### 2.1 Strata Overview

Strata operates as a co-routine with the binary it is translating, as shown in Figure 1. As the figure shows, each time Strata encounters a new PC, it first checks to see if the PC has been translated into the *fragment cache*. The fragment cache is a software instruction cache that stores portions of code that have been translated from the native binary. The fragment cache is made up of code *fragments*, which are the basic unit of translation. If Strata finds that a requested PC has not been previously translated, Strata allocates a fragment and begins translation. Once a termination condition is met, Strata emits all *trampolines* that are necessary. Trampolines are chunks of code emitted into the fragment cache to transfer control back to Strata. Most control transfer instructions (CTIs) are initially linked to trampolines (unless its target previously exists in the fragment cache). Once a CTI's target instruction becomes available in the fragment cache, the CTI is linked directly to the destination, avoiding future uses of the trampoline. This mechanism is called *Fragment Linking* and avoids significant overhead associated with returning to Strata after every fragment [3].

Strata's translation process can be overridden to implement a new SDT use. The basic Strata system includes several default behaviors that control the creation of code fragments. These default decisions can be changed based on a particular SDT use.

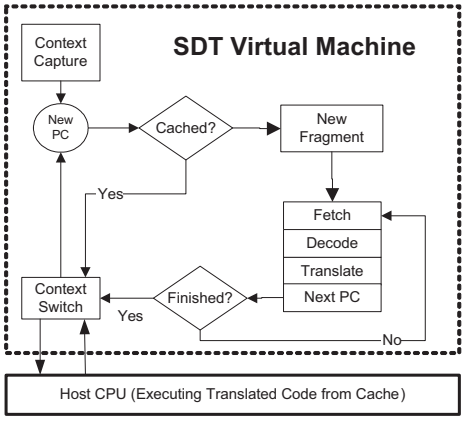


**Figure 1: Strata high-level overview.**

**Overhead Reduction**. We evaluated a number of designs to determine which are indeed the best as the default. For example, through extensive experimentation on a variety of machines, we determined when Strata should stop translating instructions into the fragment cache, how Strata should handle direct control transfer instructions, where to place code to return control from the application to Strata, how to align branch targets, and how aggressively to translate previously unexecuted functions.

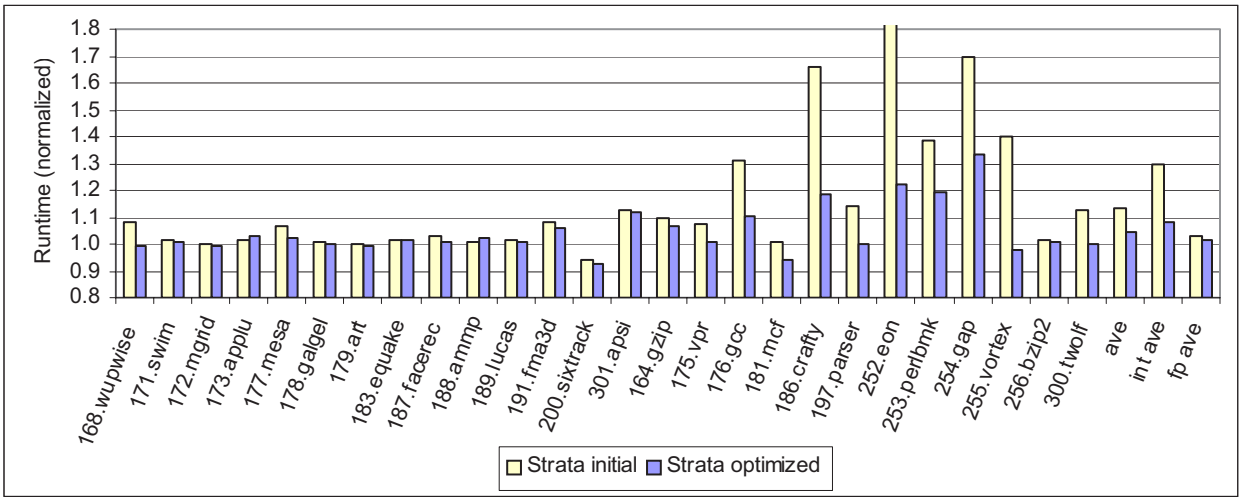


**Figure 2: Performance on Strata with initial configuration and optimized configuration for UltraSparc IIi.**

One of our most surprising findings involves the performance of partial inlining. Partial inlining is a technique in which call instructions are elided and the first portion of a function is inlined into the calling function's code. It was previously believed that this technique helped performance by eliminating unnecessary control transfer instructions. In fact, we found that partial inlining significantly degraded performance because of increased branch mispredictions. We believe the increase in mispredictions is due to the hardware return address stack being prohibited from efficient use.

**Experimental Results**. Figure 2 shows how the changes to Strata improved performance on an UltraSparc IIi. On average, we see that overhead was reduced from 13% to 4%, while on individual benchmarks we see reductions as much as 119% to 25%. Results indicate that this performance gain is similar on an AMD Athlon Opteron machine, and even more pronounced on an Intel Pentium IV Xeon with a deeper pipeline and smaller instruction cache. The largest gains result from eliminating the use of partial inlining and translating beyond conditional control transfer instructions. Previous publications can full details [17].

## 2.2 Dynamic Instrumentation

A key requirement for a dynamic optimization framework is to monitor applications as they are running. The technique used by optimizer for program monitoring is called dynamic instrumentation. Dynamic instrumentation involves inserting additional code into the program to track program properties and values. This section describes the approach to dynamic instrumentation in our dynamic optimization scheme and techniques to mitigate the associated overheads.

**Dynamic Instrumentation**. Traditional dynamic instrumentation systems need to be tailored to specific applications and platforms. For the continuous compilation framework, what is needed is a dynamic instrumentation system that is flexible enough to be used for different monitoring purposes and adaptable to different architectural platforms. We developed FIST (Flexible Instrumentation system for Software dynamic Translator) that supports the diverse instrumentation needs and platform independence needed by our dynamic optimizer.

FIST makes decisions about where and how to instrument an application based on run-time behavior. FIST is based on an event-response model that triggers information gathering when a property about the running program is satisfied. In FIST, an event occurs when a program monitor discovers that a run-time property has been satisfied. A response is an *action* taken for that event. This reactive model permits trade-offs between the cost and amount of information gathered. An example of an event is the increment of a counter and a check to see if the associated count exceeds a value. The response code can be a notification to the run-time system and a subsequent action.

To provide the flexibility that our dynamic optimizer needs, FIST uses three primitives for instrumentation: *inline-hit-always*, *hit-once*, and *hit-many.* These primitives are used to build more complex operations, and they differ in the way in which instrumentation is inserted and left in the application. Inline-hit-always is inserted directly into a basic block and never removed. Hit-once is executed outside of the program control flow and is removed immediately after being hit. Similarly, hit-many is executed outside of regular control flow and remains in the code until explicitly removed. Hit-once and hit-many intercept

control flow and change it to go out-of-line to another location.

FIST is implemented in the Strata run-time system and is currently available for SPARC platform [8]. FIST has been used in the context of several instrumentation applications including a cache simulator, a software security checker and several program profilers. Our results show the performance and memory costs of FIST are very reasonable.

**Instrumentation Optimization**. Despite the relatively low overhead of FIST, a need and opportunity exist to further reduce overheads of dynamic instrumentation. Indeed, our dynamic optimizer strives to improve the performance of programs and a low-overhead monitoring system fits the framework well. While the cost of dynamic instrumentation can be reduced on a case-by-case basis [11], what is needed are automatic techniques that systematically reduce instrumentation overheads at an algorithmic level. These techniques, which we call Instrumentation Optimizations, are analogous to compiler optimizations.

To illustrate our instrumentation optimizations, consider Figure 3 that shows three instrumentation points. An *instrumentation point* is a specific point in the program which is instrumented to invoke an *instrumentation payload* at run-time. Each instrumentation point has a probe (shown as an oval) that is code that intercepts program execution to invoke the payload (shown as a box). There are three primary sources of overheads in a dynamic instrumentation system: (1) number of probes, (2) cost of the probes, and (3) cost of the instrumentation payload. We describe three instrumentation optimizations that tackle each of these three sources of overheads.

The first optimization, called Dynamic Probe Coalescing (DPC), analyzes instrumentation points in a code region along with the intervening program code to determine if the information collected at one instrumentation point can be collected at another instrumentation point. If so, the two instrumentation points are coalesced together, thereby reducing the total number of probes executed. Note that DPC reduces the dynamic number of instrumentation points, but does not reduce the amount of information collected by the instrumentation system.

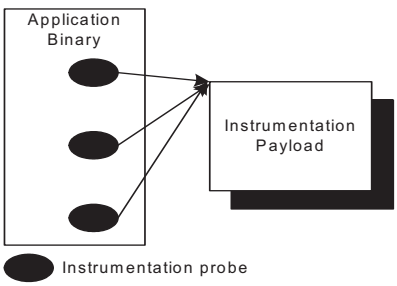


**Figure 3: Instrumentation Point**

The second optimization, called Partial Context Switch (PCS), strives to reduce the cost of an individual probe by reducing the context-switch overhead between program and instrumentation payload. PCS analyzes each instrumentation probe and the associated program code to determine the minimal number of registers that need to be saved and restored for context-switch.

The third optimization, called Partial Payload Inlining (PPI), targets the cost of an instrumentation payload by inlining the payload inside the instrumentation probe. This optimization eliminates the overhead associated with a function call and exposes further opportunities for PCS optimization.

We implemented the instrumentation optimizations in Strata-SPARC and evaluated the efficacy of the instrumentation techniques in the context of several program profilers. Table 1 shows the speedups obtained by applying the three optimizations for different program monitoring applications. The first column in Table 1 shows the name of the monitoring application; the second column gives a brief description of the application; column 3 contains the average run-time for SPEC2000 benchmarks for each monitoring application when no optimizations have been applied; and column 4 shows the same when all optimizations have been applied. The results show that the optimizations are highly effective and reduce the overheads by an average of 2.15 times. The difference in optimization effectiveness across different benchmarks and profilers are due to the fact that optimizations are more effective when there are more opportunities. Detailed results and explanations can be found in [9].

| Profilers | Description | Base | Opt. | Speedup |
|---|---|---|---|---|
| BB count | Execution count of each basic block | 487s | 218s | 2.15x |
| Path profile | Gather blocks executed along a path | 448s | 218s | 1.96x |
| Address profile | Collect load/store addresses | 589s | 219s | 2.54x |
| Value profile | Collect values used by loads and stores | 585s | 211s | 2.63x |
| Branch history | Record taken and not-taken branch history | 511s | 212s | 2.26x |
| Call-chain | Record order of function calls and returns | 712s | 585s | 1.26x |

**Table 1: Instrumentation optimization speedups.**

## 3. Model-based Optimizations

To *systematically* address the challenge of applying dynamic optimizations, we need to have a better understanding of the properties of optimizations. Our approach is to develop a general framework for studying optimization properties, in particular, profitability. Based on whether an optimization is profitable, decisions can be made about when to apply the optimization. Because of the

high cost associated with applying optimizations and experimentally evaluating profitability, we use an *analytical* approach to develop a model-based framework to predict the profitability of optimizations.

Our framework, given in Figure 4, has three types of analytic models (code, optimization and resource models) and a profitability engine that processes the models and computes the profit. The models are plug-and-play components. When new models for the code, optimizations or machine resources are needed, they can be developed and easily added into the framework.

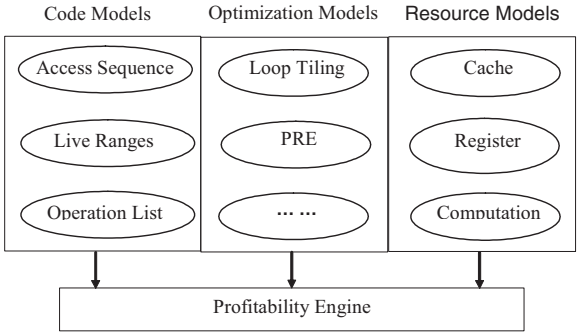


**Figure 4: Predicting Optimization Profitability.**

A code model expresses those characteristics of the code segment that are changed by an optimization and impact a machine resource. In our framework, there is a code model for each machine resource. For example, there is a register code model to express live range information because live ranges can be changed by an optimization and impact the registers. An optimization model expresses the semantics (i.e., effect) of an optimization, from which the impact of the optimization on each resource can be determined. A resource model describes the resource configuration and benefit/cost information in using the resource. The resource models are developed based on a particular platform. As part of the framework, there is a profitability engine that uses the models to predict the profit of applying an optimization.

We have developed the framework instances for predicting the profitability of scalar optimizations and loop optimizations. The machine resources that we consider include cache, registers and computation. In the next section, we will present these framework instances for loop optimizations and the experimental results to demonstrate the effectiveness of our framework. See [7] for information about the framework instances for predicting the profitability of scalar optimizations.

### 3.1 Prediction for Loop Optimizations

Data caches are designed to exploit locality, and naturally they work best for programs that have high locality. Some loop optimizations are designed to improve cache performance by transforming the code to have better locality. However, other optimizations are not designed specifically for this purpose and may negatively impact cache performance and the overall performance.

We have developed a framework instance for predicting the profitability of loop optimizations. Since loop behavior tends to dominate cache performance, we are focusing on the profitability on cache performance.

**Framework for Loop Optimizations**. The **cache code model** represents code characteristics that affect the cache. It captures several aspects of a loop nest: (1) the loop header with its lower and upper bounds and iteration step; (2) all array references and their type (includes read and writes and their affine expression); and (3) an array reference sequence that consists of all array references in a loop body in the order that they appear in the intermediate code.

A **loop optimization model** represents a loop optimization by a sequence of functions that affect the various aspects of the cache code model. For example, in loop reversal, the direction in which a loop traverses its iteration range is reversed. Our optimization models have an impact function that describes the loop header is changed to the new traversal order. We have developed optimization models for loop interchange, loop tiling, loop reversal, loop fusion, loop distribution and loop unrolling.

The **cache resource model** expresses the data cache configuration in the particular platform, including the cache size, cache block size, associativity and cache miss penalty.

The **profitability engine** takes the cache code model, loop optimization models and the cache resource model to predict the number of cache misses increased or decreased after applying an optimization.

More details and examples of framework for predicting the profitability of loop optimizations are in [6].

**Experimental Results**. To investigate the effectiveness and usefulness of our framework toward predicting the profitability of loop optimizations on cache performance, we implemented our models and tested them with several benchmark loops, including *alv, irkernel, lgsi, smsi, srsi, tfsi, tomcat3, biquad, gdevcdj, lms* and *pegwit*.

Using these benchmarks, we validated the prediction accuracy of our framework. We also showed the performance improvement of our model-based optimizations over always applying loop optimizations. Finally, we showed that our framework can also be used to select the most beneficial optimizations. The complete experimental results are presented in [6].

Table 2 gives the prediction accuracy of our framework for each benchmark and loop optimization considered. The prediction accuracies in the table are averages across a

range of trip counts for each benchmark. The trip count was varied from 50 to 200 for each benchmark to simulate different ratios of working set size to cache size and to determine whether our model can accurately reflect different loop configurations. The prediction accuracy of our framework in determining the profitability of loop optimizations is 96% on average. The prediction accuracy for loop reversal on *lgsi* is 82%. This lower prediction accuracy is because for most trip counts, the cache miss reduction of loop reversal is so small (the reduction is just one or two misses) that our model can not predict the benefit. Instead, our model does not apply loop reversal in these cases when the miss reduction is so small. Not applying reversal in this case does no harm since the relative improvement of applying reversal is minimal and can be ignored.

| Benchmark | Interchange | Tiling | Reversal |
|---|---|---|---|
| alv | 100% | 100% | 97.4% |
| irkernel | 98.7% | 100% | 93.4% |
| lgsi | 100% | 100% | 82% |
| smsi | 100% | 100% | 86.8% |
| srsi | 100% | 100% | 86.8% |
| fsi | 100% | 97.4% | 100% |
| tomcat3 | 98.7% | 92.1% | 93.4% |
| biquad_N | 89.5% | 88.2% | 100% |
| gdevcdj | 100% | 100% | 97.4% |
| lms | 97.4% | 100% | 94.7% |
| pegwit | 100% | 100% | 81.6% |

**Table 2: Loop Optimization Prediction Acccuracy.**

# 4. Dynamic Optimization

Although dynamic optimization has shown promising potential, its effectiveness and efficiency have not been fully understood, which is especially true for native-to-native optimizers. For example, Dynamo does not achieve improvement on some programs [13], and DynamoRio results in a slowdown (12% on average) over SPEC 2000 integer benchmarks [14]. Further exploration is needed to extend the applicability of dynamic optimization. The goal of this project is to develop a dynamic optimization scheme that is effective and efficient.

## 4.1 Dynamic Optimization

We implemented a framework for dynamic optimization with a set of optimizations implemented in Strata. Similar to Dynamo, the optimizer monitors program execution, detects a frequently executed path, forms it into a superblock trace, optimizes it, and uses the optimized trace for future executions. Currently, the optimizer performs a few scalar optimizations, namely, constant propagation, copy propagation, constant folding, and algebraic simplification. It removes dead code, partially dead code, and some redundant code. Moreover, when a trace inlines the entire body of a routine, it can remove the call and return since they are no longer needed. Due to the simplicity of the control flow of a superblock trace, these optimizations are performed rapidly in a forward pass and a backward pass.

We find that native-to-native optimization needs special care to determine an optimization opportunity, compared to traditional source-to-native compilation. Native-to-native optimization accepts a native code segment as input. A seemingly poor native instruction sequence may be produced by a poorly performing compiler; it may also be generated out of necessity (e.g., hardware constraints, such as the range of an immediate number). The former case is a *true optimization opportunity* because the code can be improved, while the latter is a *false optimization opportunity* because the code poorness is necessary. A dynamic native-to-native optimizer should detect false optimization opportunities and avoid wasting resources on optimizing them. Source-to-native compilation, on the contrary, typically performs optimization on an intermediate representation before code generation. Therefore, traditional compilers seldom need to handle false optimization opportunities.

## 4.2 Trace Quality

Trace quality dramatically affects the result of dynamic optimization. Many opportunities for dynamic optimization result when a trace is formed into a joint-free superblock, where a variable in the original program may become a constant number, and a partial redundancy may become full redundant. Longer traces contain more optimization opportunities.

Traces should also follow execution flow. The benefits of optimization are realized only when optimized code executes. If execution goes off a trace in the middle too often, the time to optimize the code that is not executed is improvident. In addition, infrequently executed code in traces may degrade cache performance.

Many dynamic optimization systems employ the Next-Executing Tail (NET) technique to select traces [15]. Our experiments indicate that 40.2% of NET traces consist of a single basic block, and Hiniker et al. showed the average size of a NET trace is 14.8 instructions [16]. In addition, our experiments show that 49.9% of execution does not follows NET traces from the trace head to the tail, going off traces in the middle. We believe there is much opportunity to improve the trace quality over the NET technique and we are investigating new trace selection algorithms.

## 4.3 Static Planning for Dynamic Optimization

A dynamic optimization system has tighter resource constraints than static compilers. For instance, the code win-

dow a dynamic optimizer sees is smaller. Nevertheless, some sophisticated transformations naturally need context information of the code that is being optimized. We propose *static planning* to address this challenge. A static plan contains program information that is required for transformation but is expensive or impossible to compute at run-time (e.g., data flow). As our experience indicates that a dynamic optimizer spends much time checking the applicability of a transformation, static planning can improve the efficiency by computing the required information beforehand. To increase the effectiveness of dynamic optimization, a static plan also suggests transformations for a dynamic optimizer to apply at run-time, including what optimizations to perform, configuration of the optimizations, and the order to apply them. A model-based optimization method has been developed to study the properties of optimizations (previously discussed in Section 3). This analytical model-based approach is well suited for the static planning scheme—generating dynamic optimization plans with a high confidence of profitability.

We are currently investigating the benefits, feasibility, and formats of static planning for dynamic optimization.

## 5. Debugging

Debugging is the process of eliminating bugs by pausing and stepping through execution and inspecting and modifying values. Debugging plays an important role in software development. Dynamic translation in our framework introduces novel challenges to debugging programs. In particular, since program code is generated at run-time, the static debug information associated with the program becomes inconsistent with the dynamically changing program. Transformation of translated code, such as overhead reduction techniques and dynamic optimizations, change the number of instances and order of statements, further complicating the debugability of the program. Finally, dynamic instrumentation performed as part of program monitoring inserts additional code into the program that has no relation to the executing application, but still executes with the application. Such instrumented code must be hidden from a debug user who is unaware of modifications to the program at run-time.

Our goal is to provide capabilities to users that allow transparent debugging of dynamically translated programs. Debugging is *transparent* in that a debugger user is kept completely unaware of dynamic code modifications performed our system. Another goal is to keep the debugging techniques independent of the target platform. This way, the same debugging framework suffices for different platforms supported by our dynamic optimizer. Finally, one of our goals is to provide the same debugging facilities and commands to users as existing debuggers do.

To achieve our goals, we propose a debug architecture as shown in Figure 5. The debug architecture has three components: (1) a SDT system, which is the run-time system of our dynamic optimizer (2) a native debugger that is being extended to support dynamically translated programs, such as gdb, and; (3) a debug engine. The debug engine is a critical component that generates debug information at run-time. This information is then used to hide dynamic code generation and code modification from the native debugger (and debug user).

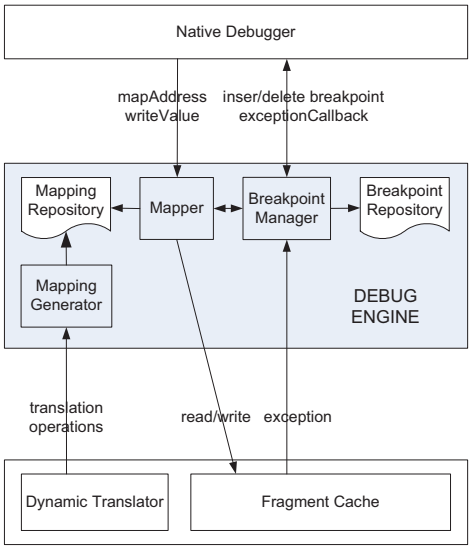


**Figure 5: Debug Architecture**

The debug engine consists of three components: a mapping generator, a mapper, and a breakpoint manager, and two repositories, the mapping repository and the breakpoint repository. The components and repositories are shown in Figure 5. The mapping generator computes debug information, consisting of dynamic debug mappings that relate source program locations to translated program locations and the vice-versa. It uses information provided by the SDT system for generating and updating mappings. The mappings, tuples relating untranslated program locations with translated locations, are stored in the mapping repository. The mapper uses the debug information from the generator to map untranslated and translated code. The output of the mappings can be used by either the native debugger or the SDT system.

The breakpoint manager keeps track of all active breakpoints (and watchpoints) for the executing program in the breakpoint repository. The native debugger communicates information about the breakpoints to the breakpoint manager. The breakpoint manager is responsible for inserting breakpoints in translated code. When breakpoints are hit in the translated code, the breakpoint manager is notified.

To understand the flow of information through the debug engine, consider a typical debug session when a user tries to insert a breakpoint at a source location. The native debugger computes the corresponding untranslated program location and invokes the breakpoint manager. The breakpoint manager consults the mapper to determine the corresponding translated locations and inserts breakpoints at each of these locations. The breakpoint manager also saves the breakpoint information in the breakpoint repository. When a breakpoint is hit in a translated location, an appropriate untranslated binary location is reported to the native debugger. Debug commands such as single-stepping, inspection of stack frames, inspection and modification of data values are also supported through the debug engine. Currently, the debug engine does not provide debugging techniques for dynamic optimizations.

We implemented the debug architecture in a debugger called Tdb, using the Strata as the SDT system and gdb as the native debugger. We verified the correctness of Tdb by comparing results obtained by Tdb while debugging translated programs and comparing them to untranslated programs being debugged using Gdb [12]. Detailed performance results for Tdb are described in [10].

## 6. Conclusion

To realize next-generation performance requires next-generation optimization technology. Because of the serious limitations of current optimization techniques, our research is focusing on dynamic optimizations. Our system will use both static and dynamic information by forming plans at compile time for online optimization. The runtime system, based on a software dynamic translator, continuously monitors the running application and applies optimizations according to both the compilation-time-generated plans and traditional dynamic optimization techniques.

Extensive progress has been made in developing this system. First, we studied the causes of Strata's overhead and found methods to dramatically reduce overhead, to as little as 4% extra execution time on average. Progress has been made in ways to efficiently monitor the running program via the FIST instrumentation system and INSOP system for optimizing code to monitor the application. INSOP can reduce the cost of monitoring a program by as much as 215%. To widely deploy dynamic optimizers, users and compiler writers must be able to debug applications. Consequently, we developed and implemented a method for debugging programs running within our runtime system. Further progress has been made on predicting the performance impact of applying optimizations using hardware models. Results indicate that the prediction quality is exceptional, over 85% accuracy in most cases. Lastly, significant progress has been made in applying dynamic optimization.

## 7. References


[1] Bacon, D. F., Graham, S. L., and Sharp, O. J. 1994. Compiler transformations for high-performance computing. ACM Comput. Surv. 26, 4 (Dec. 1994), 345-420.

[2] 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*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*, September 2001.

[4] K. McKinley, S. Carr and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, Vol. 18, No.4, July 1996.

[5] Michael D. Smith and Glenn Holloway. An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization. URL: http://www.eecs.harvard.edu/hube/software/nci/overview.html

[6] M. Zhao, B. Childers and M.L. Soffa. Predicting the Impact of Optimizations for Embedded Systems. ACM Conference on Languages, Compilers, and Tools for Embedded Systems, June 2003.

[7] M. Zhao, B. Childers and M.L. Soffa. A Model-based Framework: an Approach for Profit-driven Optimization. International Symposium on Code Generation and Optimization, March 2005.

[8] N. Kumar, J. Misurda, B. R. Childers and M. L. Soffa, "Instrumentation in software dynamic translators for self-managed systems", ACM Workshop on Self-Managing Systems, 2004.

[9] N. Kumar, B. R. Childers and M. L. Soffa, "Low overhead program monitoring and profiling", ACM Workshop on Program Analysis for Software Tools and Engineering, 2005.

[10] N. Kumar, B. R. Childers and M. L. Soffa, "Tdb: A source-level debugger for dynamically translated programs", ACM Conf. on Automated and Analysis-Driven Debugging, 2005.

[11] 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", ACM Conference on Programming Language Design and Implementation, 2005.

[12] R. M. Stallman and R. H. Pesch, "Using GDB: A guide to the GNU source-level debugger", GDB v4.0, Free Software Foundation, Cambridge, MA, 1991.

[13] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. Conf. on Programming Language Design and Implementation (PLDI '00). June 2000.

[14] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. Intl. Symp. on Code Generation and Optimization (CGO '03). March 2003.

[15] E. Duesterwald and V. Bala. Software Profiling for Hot Path Prediction: Less is More. Proc. of 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. November 2000.

[16] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving Region Selection in Dynamic Optimization Systems. Intl. Symp. on Microarchitecture (MICRO-38). November 2005.

[17] Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, Bruce R. Childers. Evaluating Fragment Construction Policies for SDT Systems. Submitted for publication to Second International Conference on Virtual Execution Environments, June 2006.