Efficient, Compiler-Aided Dynamic Optimization

David Hodgdon

2nd November 2004

Abstract

While dynamic optimizers perform well when improving code generated with little compile-time optimization (-O0, -O1), optimizing code with aggressive compile-time optimizations yields little performance increase, often increasing run-time due to the overhead of identifying and building hot traces. As dynamic optimizers mostly use information also available to the compiler via profiling, their inability to improve already optimized code is not surprising.

In order to improve aggressively optimized binaries, the dynamic optimizer must utilize information that is not available to the compiler even with profile feedback. The proposed dynamic optimizer will utilize run-time data values and data object liveness information to expose additional optimization opportunities in traces. By assuming certain values remain constant or objects remain allocated, the trace must be removed when these precondition assumption become invalid. The compiler can reduce analysis overhead in the dynamic optimizer by determining which objects have long lifetimes and which objects have relatively constant values. The compiler can also insert instrumentation calling the dynamic optimizer in code regions which potentially violate trace assumptions, allowing the optimizer to sleep reducing overhead. To address redundant profiling and trace optimization in distributed environments, a server thread maintains a trace library and coordinates distributed path profiling for trusted threads of the same program.

1 Introduction and Related Work

1.1 Hardware Techniques

Computer architects have developed many hardware-based, run-time methods to improve performance. Out-of-order execution[15][16][27] allows the processor to execute any ready instruction from a fixed instruction window. Branch prediction and speculative execution prevent costly pipeline stalls. The trace cache[9][23] aids fetching by storing instructions in their dynamic execution order. Modern microprocessors implement many hardware-based, run-time optimizations.

1.2 Software Dynamic Optimization

Software-based dynamic optimization systems have only recently been explored as an alternative means to exploit run-time behavior. A few systems of this kind have been implemented for research purposes, but none have become as ubiquitous as their hardware-based counterparts. First, the dynamic optimizer observes run-time characteristics to determine hot traces, frequently executed control flow paths much like superblocks[11] each with one entrance and multiple exits. These traces are able to span procedure boundaries, even into shared libraries. Several low overhead optimizations may be performed before the trace is placed into a software managed code cache. Subsequent runs of this hot control flow path will execute as straight-lined code from the code cache. Since hot traces jump from one to another are linked together, the execution will rarely exit the code cache. Forming and managing hot traces adds overhead; however, the performance gain of running straight-line code generally outweighs the time spent building traces[4]. Furthermore, when simple optimizations are applied to traces, performance gains are achieved[3].

Dynamo[3] is a dynamic optimizer for a PA-RISC machine developed at Hewlett-Packard Laboratories. It acts as a native interpreter which allows it to observe run-time behavior without instrumentation. Dynamo selects potential trace heads and monitors their execution counts. When the execution count of a potential trace head goes above a hot threshold, a trace is formed. Optimizations such as elimination of redundant branches, constant propagation, and strength reduction are performed before the traces are placed into a trace cache. DynamoRIO[4], a system created during a collaboration between Hewlett-Packard and MIT, is an x86 system that is based on Dynamo. Instead of interpreting the x86 instructions, run-time information is achieved by executing instrumented basic blocks from a basic block cache. DynamoRIO also exports a rich API for a user to instrument the basic blocks and traces.

Like DynamoRIO, DELI[7] and Strata[8] export an API enabling custom optimizations. Wiggins/Redstone[6] uses performance counters on the Alpha to build traces. Ispike[19] and ADORE[18] utilizes performance counters available on the Intel IA-64 architecture to perform efficient edge profiling, reducing the overhead of a dynamic optimization.

Path profiling provides superior information to dynamic optimizers though at a higher cost. Targeted Path Profiling [14] and Practical Path Profiling (unpublished) attempt to reduce the overhead associated with path profiling.

The Vulcan [25] static/dynamic binary transformation tool transforms x86, IA-64, or MSIL code into an abstract representation before transforming it back to x86, IA-64, or MSIL code. While in the abstract representation, code can easily be modified utilizing an extensive API. Mojo[5] is a dynamic optimizer that utilizes the Vulcan API to targets large, multi-threaded Microsoft Windows™ applications.

PIN [12] is Intel's cross-platform (IA-32, IA-32E, IA-64, and ARM) instrumentation tool (with planned Microsoft Windows™ support). The ROGUE dynamic optimization system [22] is a component of PIN which exports a rich API for dynamic optimization. ROGUE is inherently portable, since it is based on PIN.

1.3 Dynamic Compilation

Dynamic-compilation systems provide similar code enhancements by delay some compilation until run-time. In the extreme, Just-In-Time (JIT) compilers (such as [24] for the Java language) compile all code at run-time, while others such as [17, 21, 10] compile only select regions of code but require programmer annotations. Calpa [20] is a tool that automates these annotations. Like the proposed research, these systems perform run-time specialization; however, the proposed research differs in that all code is precompiled, and reoptimization (binary-to-binary, opposed to source-to-binary) only occurs on hot traces, reducing the overhead due to recompiling unimportant code at run-time.

1.4 Shortcomings of Previous Work

Software dynamic optimizers have never become widely accepted for four reasons.

1. **Monitoring overhead**: The overhead associated with interpretation or running instrumented code hides some or all of the performance benefits that a dynamic optimizer provides.

2. **Poor decisions**: Since the dynamic optimizer is limited in analysis time, it is prone to making poor decisions. For example, due to the inaccuracies of edge profiling (vs path profiling), "hot

traces" are built which are never fully executed, resulting in wasted time and unnecessary code bloat (i-cache performance).

3. **Aggressive Optimization**: Since dynamic optimizers perform similar optimizations to compilers armed with profile-feedback, such as superblock/trace formation (and optimizations exploiting the additional ILP) and prefetching, they cannot achieve significant performance improvements over aggressively optimized code.

4. **Verifiability**: For many years companies have shipped programs with little or no compiler optimization. When an optimizing compiler contains a bug (common), program bugs can be created from correct code. Luckily, these bugs can be fixed and compiler optimizations can be held to higher standards of formal verification. However, even completely correct optimizations can reveal bugs in the program source which were hidden without optimization. Dynamic optimizers face a worse problem in non-repeatable bugs. Not only do random variables and user input influence control flow paths and data values but also compiler optimizations.

Dynamic compilers have never become widely accepted for four reasons.

1. **Overhead**: Allowing multiple specializations of a single code region results in significant overhead. First, a dispatcher must be used to execute the correct code region (instead of direct program flow). Secondly, maintaining a large number of region specializations reduces instruction cache performance.

2. **Annotations**: Most systems require the programmer to manual identify code regions to optimize or runtime constants with which to specialize.

3. **Source Code Revealed**: Commercial software vendors are unwilling to ship their programs with source code so that it can be dynamically compiled on customer's home computers.

4. **Verifiability**: Like compilers and dynamic optimizers, dynamic compilers can contain or reveal bugs.

1.5 Proposed Solutions

Hardware Path Profiling Path profiling allows the dynamic optimizer to produce high-quality traces. Current research indicates that hardware performance counters can be used to efficiently generate path profiles.

Compiler Assisted Dynamic Optimization By allowing the compiler to perform the most costly analysis, the dynamic optimizer can reduce optimization overhead. The compiler can also pass down high-level information to the dynamic optimizer which is otherwise lost in the translation process, while not providing the source code.

Aggressive Optimizations The dynamic optimizer can build specialized traces based on runtime constants (and near-constants) and variable liveness. These specializations expose optimization opportunities such as aggressive memory transformations, complete loop unrolling, dead code elimination, code motion, and constant propagation and folding.

Figure 1: 183.Equake Example

```
0: #define PI 3.141592653589793238
1: double phi2(double t) {
2:   if (t <= Exc.t0)
3:     return 2.0 * PI / Exc.t0 / Exc.t0 * sin(2.0 * PI * t / Exc.t0);
4:   else     return 0.0;
5: }
```

Distributing Work A server thread maintains a trace library and coordinates distributed path profiling for trusted threads of the same program. Maintaining a trace library reduces the overhead associated with optimizing the same trace multiple times. Distributing path profiling reduces the overhead of path profiling, while potentially increasing the coverage.

Verifiability Even after all optimizations are verified, they can still expose bugs in the original program. To verify correct behavior of a typical dynamic optimizer's interaction with a potentially buggy program, one must run the program will all possible inputs times all possible trace building combinations. However, the approach is still not complete due to inconsistencies in shared library calls (such as different addresses or alignment of memory allocation calls). Therefore, performing an exhaustive search for possible conflicts is impossible. Instead, dynamic optimizers must maintain the semantics of common debugging tools such as assertions and run-time array bounds and pointer checking. To this end, the proposed dynamic optimizer can optimize array bounds and pointer checking to make it more feasible.

2 New Dynamic Optimization Opportunities

2.1 Run-time Near-Constants

Many optimization opportunities are lost at compilation time due to the unknown value of run-time constants. These constants are often defined in a configuration file, just out of reach of static compiler optimizations. A compiler can often determine that a value is a run-time constant, but cannot determine the value of it; therefore, the compiler can pass this information to the dynamic optimizer. Once the value is set, the optimizer can build a trace utilizing the new knowledge with constant propagation, constant folding, strength reduction, branch folding, complete loop unrolling (if the constant is a small loop counter), etc.

However, in some cases, the compiler cannot absolutely determine that the variable is only set once. In other cases, the near-constant variable receives new values infrequency, for example, on large-scale phase changes. Since the variable remains constant for long periods of execution, it is desirable to optimize traces based on these values. The compiler can insert instrumentation which calls the dynamic optimizer when the run-time near-constant is set. No urgency exists on the first call, the value is simple provided to the dynamic optimizer; however, on future calls, the optimizer must invalidate any trace that it has been built utilizing the value of the run-time near-constant.

In the 183.Equake benchmark in SPEC2000, there are three functions called `phi0`, `phi1`, and `phi2`, which each account for between 3% and 4% of execution (when they are not inlined). `Exc.t0` is a run-time constant set in `main()`. Figure 1 presents the code for `phi2` (slightly cleaned-up). The dynamic optimizer could remove a load instruction (the load of `Exc.t0`, line 2) and two floating

Figure 2: Early Load Example

```
0: double* costable;
1: double cos_law(double a, double b, int C) {
2:   int val = a*a+b*b;
3:   if(C%180!=90)  val -= 2*a*b*costable[C%360];
4:   return sqrt(val);
5: }
6: double pythag(double a, double b) { return cos_law(a,b,90);}
```

point divides (the first two occurrences of `Exc.t0`, line 3, since now `2.0 * PI / Exc.t0 / Exc.t0` is a single constant. The load of `Exc.t0` would almost never miss in the L1 data cache; however, removing it is still beneficial. Removing two floating point loads should speed up the function significantly.

2.2 Aggressive Memory Transformations

Due to the inherently long latency nature of load instructions, optimizing compilers attempt to schedule loads as soon as possible, placing independent instructions between the load and its first use. Two problems impede early scheduling of loads (besides data dependencies and available registers). First, a load cannot be promoted above a store, unless it is known that that store and load have non-overlapping effective addresses, determined using pointer analysis. Second, a load cannot be moved into a basic block where it could result in a fault which could not occur in the original layout.

To systematically avoid such faults, a load can be promoted from block A to B only if the compiler can determine that the load address points to a valid object in block B. In C [1] there are three lifetimes classes for objects: static (live for entire program), automatic (live for scope), and allocated (live from allocation to free or reallocation). If the compiler can determine that the lifetime of an object extends over B as well, the load can be promoted. While the analysis for static and automatic objects is straight forward, determining liveness of an allocated object proves difficult. Typically compilers cannot determine the liveness of an allocated object, which make up a large portion of total objects (used in all dynamic data structures).

The compiler can insert instrumentation near memory management calls to monitor the liveness of allocated objects. When the dynamic optimizer builds a trace, it checks the current liveness of objects whose associated loads could be promoted. If an object is currently live, it can be promoted; however, the trace must be invalidated if the object freed or reallocated. Therefore, the compiler inserts callbacks to the dynamic optimizer near free or reallocate instructions.

Figure 2 presents an example with a law of cosines function and a Pythagorean theorem function. To increase performance it would be best to schedule the load of `costable` (line 3) as early as possible. However, it is unknown if the object to which `costable` points is valid. The object could certainly be unallocated if `pythag` was the function being called and the condition in line 3 always failed. The dynamic optimizer can check to see if `costable` has been allocated. If so, the trace can be preconditioned on `costable`'s value (the pointer) and the object to which `costable` points remaining allocated, allowing the load to be placed as early as possible, hiding memory latency.

Figure 3: Bounds Checking Example

```
0: int sum=0;
1: for(i=0;i<rt_const;i++) {
2:   object *obj = bc_find_object(a);
3:   if(obj && obj->base <= &a[i] && &a[i]<obj->extent)
4:     sum += a[i];
5:   else
6:     bc_error();
7: }
```

2.3 Run-time Bounds and Pointer Checking

Because of the difficulty of finding pointer bugs and the security problems related to running over the end of an array, run-time bounds and pointer checking has become very popular. Many languages already perform these run-time checks. Because C allows pointer arithmetic and is weakly typed, implementing these checking in C is difficult and results in significant overhead. That said, C run-time bounds and pointer checking is available in the Alpha cc compiler and gcc (via a popular patch) as well as a number of research systems [13, 26, 2].

By leveraging knowledge of run-time constants and object liveness, the dynamic optimizer can identify checks which are unnecessary in hot paths. Removing these checks in frequently executed code regions should significantly reduce overhead, while maintaining the correctness of the checks.

Figure 3 (Syntax and example borrowed from [13]) displays example code summing the first `rt_const` elements of an array with added bounds checking code. Aggressive compiler analysis reveals that line 2 could be pulled out of the loop. Furthermore line 3, 5, and 6 could be pulled, performing only a single check using the known bounds. The dynamic optimizer, however, could remove the bounds checking completely since it knows the object to which `a` points, the liveness and size of `a`, and the value of `rt_const` (assuming `rt_const` is a run-time constant).

3 Reducing Overhead in Distributed Environments

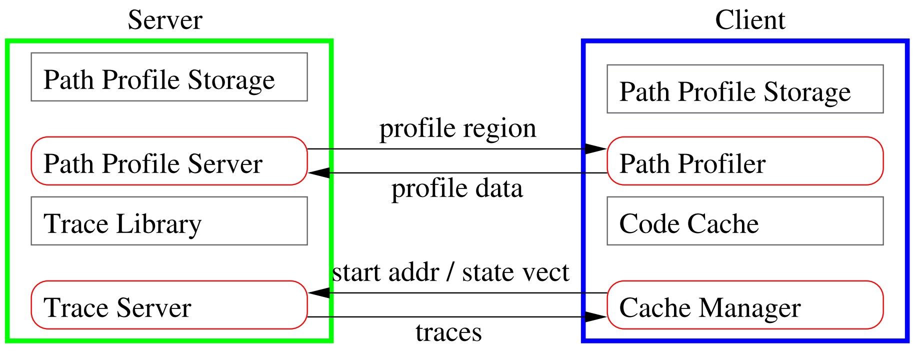
In scientific computing and server environments, where many threads of a program execute together for long periods of time, identical or nearly identical traces can be built and optimized multiple times in each thread. Also resources of each thread are used to build similar path profiles. The proposed dynamic optimizer utilizes a dedicated server thread to coordinate the sharing of traces and path profiles between client threads. Figure 4 provides an overview of the client-server interactions.

3.1 Shared Trace Libraries

In order to reuse traces, the proposed server thread will maintain a trace library contributed to by trusted client threads.

To create traces that can be used by another thread (possibly running on a different machine), the position dependent assignments which occur at load-time must be reversed. Additionally, control flow targets must be independent of other traces running on the machine, so targets are converted to point back to the original code. The trace and its preconditions are added to a buffer to be sent to the server thread.

Figure 4: Client-Server Interaction



A request for a trace in the library consists of a starting address combined with a state vector (whose elements are determined per region by the compiler). Because the overhead associated with requesting a trace is high (especially if no trace exists in the library), the thread may decide to generate the trace itself.

The server thread stores incoming traces in lists chosen by a hash-table keyed on the starting address. When servicing a request the server compares the state transmitted in the request to the preconditions of the traces (matching the starting address) in the library. The server can send back multiple connectible traces (as a prefetch mechanism of sorts).

3.2 Shared Path Profiling

In order to reduce the overhead and increase the coverage of path profiling, the server can distribute the profiling between clients. Clients are issued overlapping portions of code on which to perform path profiling and report back to the server. By analyzing differences in overlapping path profiles, the server is able to construct a specialized path profile for each client.

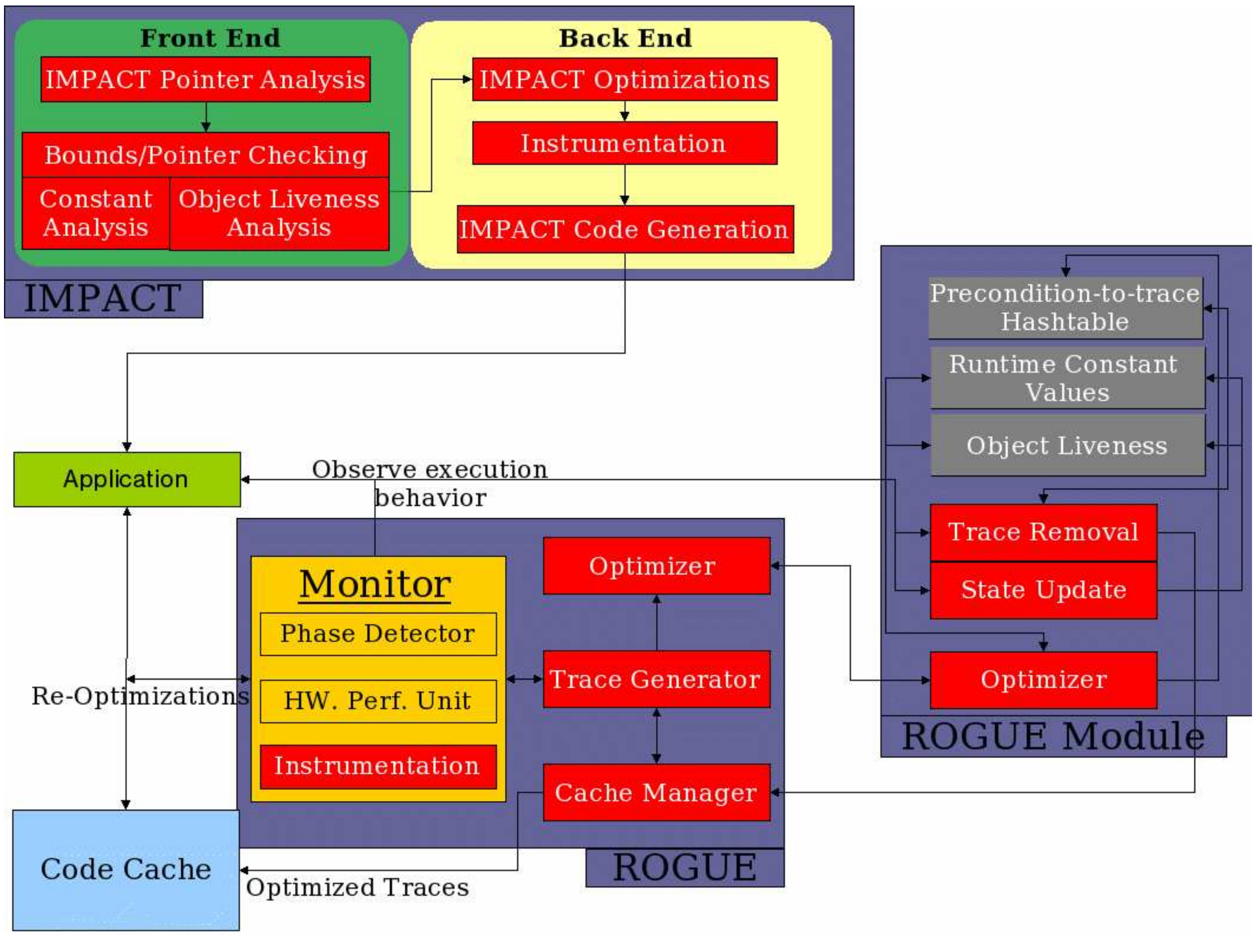
4 Research Framework

Figure 5 (ROGUE portion adapted from [22]) is a high level overview of the three components of the project: the IMPACT compiler, the ROGUE dynamic optimizer, and the preconditioned trace ROGUE plug-in module.

Compilation The IMPACT research compiler implements aggressive pointer analysis. A front-end module will be added for bounds checking and identification of constants, near-constants, and long-lived objects. The back-end module will add instrumentation (on assignment to run-time constants and memory management) which calls the dynamic optimizer and will be responsible for transferring the front-end module's analysis to the dynamic optimizer.

Dynamic Optimization Environment Using the ROGUE API, I can build a module which ROGUE will call to perform the new trace optimizations. Additionally, the module can call on the cache manager to remove a trace (once its preconditions have been invalidated).

Figure 5: High Level Outline



References

[1] ANSI. Programming languages — C. Tech. Rep. ISO/IEC 9899:1999, ANSI, 1999.

[2] Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (1994), ACM Press, pp. 290–301.

[3] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation* (June 2000), pp. 1–12.

[4] Bruening, D., Garnett, T., and Amarasinghe, S. An infrastructure for adaptive dynamic optimization. In *1st Annual International Symposium on Code Generation and Optimization (CGO-03)* (March 2003).

[5] Chen, W.-K., Lerner, S., Chaiken, R., and Gillies, D. M. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)* (December 2000).

[6] Deaver, D., Gorton, R., and Rubin, N. Wiggins/redstone: An on-line program specializer. In *Hot Chips 11* (August 1999).

[7] Desoli, G., Mateev, N., Duesterwald, E., Faraboschi, P., and Fisher, J. A. DELI: A new run-time control point. In *35th Annual International Symposium on Microarchitecture* (December 2003).

[8] Ebcioglu, K., Altman, E., Gschwind, M., and Sathaye, S. Dynamic binary translation and optimization. In *IEEE 2001Workshop on Binary Translation* (July 2001).

[9] Friendly, D., Patel, S., and Patt, Y. Putting the fill unit to work: dynamic optimizations for trace cache microprocesors. In *In Proceedings of the 31st Int'l Symposium on Microarchitecture (MICRO-31)* (1998), pp. 173–181.

[10] Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science 248*, 1–2 (2000), 147–199.

[11] Hwu, W. W., Mahlke, S. A., Chen, W. Y., Chang, P. P., Warter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., Kiyohara, T., Haab, G. E., Holm, J. G., and Lavery, D. M. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing 7*, 1 (January 1993), 229–248.

[12] Intel. *Pin User Manual*, November 2003.

[13] Jones, R. W. M., and Kelly, P. H. J. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging* (1997), pp. 13–26.

[14] Joshi, R., Bond, M. D., and Zilles, C. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *International Symposium on Code Generation and Optimization* (March 2004), p. 239.

[15] Keller, J. The 21264: a superscalar alpha processor with out-of-order execution. In *9th Annual Microprocessor Forum* (1996).

[16] Kumar, A. The HP PA-8000 RISC CPU: a high performance out-of-order processor. In *In Proceedings of Hot Chips VIII* (1996).

[17] Lee, P., and Leone, M. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation* (1996), pp. 137–148.

[18] Lu, J., Chen, H., Yew, P.-C., and Hsu, W. C. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism 6* (2004).

[19] Luk, C.-K., Muth, R., Patil, H., Cohn, R., and Lowney, G. Ispike: A post-link optimizer for the intel itanium architecture. In *2st Annual International Symposium on Code Generation and Optimization (CGO-04)* (March 2004).

[20] Mock, M., Chambers, C., and Eggers, S. J. Calpa: a tool for automating selective dynamic compilation. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (2000), ACM Press, pp. 291–302.

[21] Noel, F., Hornof, L., Consel, C., and Lawall, J. L. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages* (1998), IEEE Computer Society, p. 132.

[22] Reddi, V. J. Rogue: Dynamic optimization. In *ASPLOS XI Tutorial Call Software Instrumentation as a Tool for Architecture and Compiler Research* (October 2004).

[23] Rotenberg, E., Bennett, S., and Smith, J. E. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture* (December 1996), pp. 24–34.

[24] Sreedhar, V. C., Burke, M., and Choi, J.-D. A framework for interprocedural optimization in the presence of dynamic class loading. *ACM SIGPLAN Notices 35*, 5 (2000), 196–207.

[25] Srivastava, A., Edwards, A., and Vo, H. Vulcan: Binary Transformation in a Distributed Environment. Tech. Rep. MSR-TR-2001-50, 2001.

[26] Steffen, J. L. Adding run-time checking to the portable c compiler. *Software — Practice and Experience 22*, 4 (1992), 305–316.

[27] Tomasulo, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development 11* (January 1967), 25–33.