Dynamo: A Transparent Dynamic Optimization System

Vasanth Bala
vas@hpl.hp.com

Evelyn Duesterwald
duester@hpl.hp.com

Sanjeev Banerjia*
sbanerjia@incert.com

Hewlett-Packard Labs 1 Main Street,
Cambridge, MA 02142

www.hpl.hp.com/cambridge/projects/Dynamo

Abstract

We describe the design and implementation of Dynamo, a software dynamic optimization system that is capable of transparently improving the performance of a native instruction stream as it executes on the processor. The input native instruction stream to Dynamo can be dynamically generated (by a JIT for example), or it can come from the execution of a statically compiled native binary. This paper evaluates the Dynamo system in the latter, more challenging situation, in order to emphasize the limits, rather than the potential, of the system. Our experiments demonstrate that even statically optimized native binaries can be accelerated Dynamo, and often by a significant degree. For example, the average performance of –O optimized SpecInt95 benchmark binaries created by the HP product C compiler is improved to a level comparable to their –O4 optimized version running without Dynamo. Dynamo achieves this by focusing its efforts on optimization opportunities that tend to manifest only at runtime, and hence opportunities that might be difficult for a static compiler to exploit. Dynamo's operation is transparent in the sense that it does not depend on any user annotations or binary instrumentation, and does not require multiple runs, or any special compiler, operating system or hardware support. The Dynamo prototype presented here is a realistic implementation running on an HP PA-8000 workstation under the HPUX 10.20 operating system.

1. Introduction

Recent trends in software and hardware technologies appear to be moving in directions that are making traditional performance delivery mechanisms less effective. The use of object-oriented languages and techniques in modern software development has resulted in a greater degree of delayed binding, limiting the size of the scope available for static compiler analysis. Shrink-wrapped software is being shipped as a collection of DLLs rather than a single monolithic executable, making whole-program optimization at static compile-time virtually impossible. Even in cases where powerful static compiler optimizations can be applied, computer system vendors have to rely on the ISV (independent software vendor) to enable them. This puts computer system vendors in the uncomfortable position of not being able to control the very keys that unlock the performance potential of their own machines. More

recently, the use of dynamic code generation environments (like Java JITs and dynamic binary translators) makes the applicability of heavyweight static compiler optimization techniques impractical. Meanwhile, on the hardware side, technology is moving toward offloading more complexity from the hardware logic to the software compiler, as evidenced by the CISC to RISC to VLIW progression.

The problem with this trend is that the static compiler is taking on an increasingly greater performance burden while the obstacles to traditional static compiler analysis are continuing to increase. This will inevitably lead to either very complex compiler software that provides only modest performance gains on general-purpose applications, or highly customized compilers that are tailored for very narrow classes of applications.

The Dynamo project was started in 1996 to investigate a technology that can complement the static compiler's traditional strength as a static performance improvement tool with a novel dynamic performance improvement capability [3]. In contrast to the static compiler, Dynamo offers a client-side performance delivery mechanism that allows computer system vendors to provide some degree of machine-specific performance without the ISV's involvement.

Dynamo is a *dynamic optimization* system (i.e., the input is an executing *native* instruction stream), implemented entirely in software. Its operation is transparent: no preparatory compiler phase or programmer assistance is required, and even legacy native binaries can be dynamically optimized by Dynamo. Because Dynamo operates at runtime, it has to focus its optimization effort very carefully. Its optimizations have to not only improve the executing native program, but also recoup the overhead of Dynamo's own operation.

The input native instruction stream to Dynamo can come from a statically prepared binary created by a traditional optimizing compiler, or it can be dynamically generated by an application such as a JIT. Clearly, the runtime performance opportunities available for Dynamo can vary significantly depending on the source of this input native instruction stream. The experiments reported in this paper only discuss the operation of Dynamo in the more challenging situation of accelerating the execution of a statically optimized native binary. The performance data presented here thus serve as an indicator of the limits of the Dynamo system, rather than its potential. The data demonstrates that even in this extreme test case, Dynamo manages to speedup many applications, and comes close to breaking even in the worst case.

Section 1 gives an overview of how Dynamo works. The following sections highlight several key innovations of the Dynamo system. Section 2 describes Dynamo's startup mechanism, Section 4 gives an overview of the ==hot code selection,== optimization and code generation process, Section 5 describes how different optimized code snippets are linked together, Section 6 describes how the storage containing the dynamically optimized

---

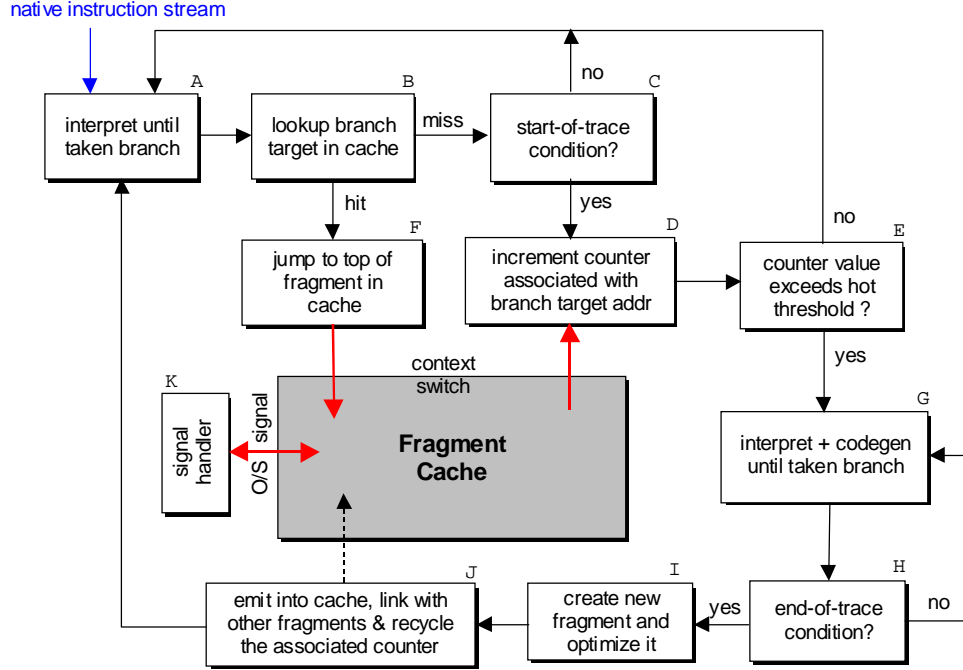*The author is presently with InCert Corporation, Cambridge, MA.

Figure 1. How Dynamo works

code is managed, and Section 7 describes signal handling. Finally, Section 8 summarizes the experimental data to evaluate Dynamo's performance. Dynamo is a complex system that took several years to engineer. This paper only provides an overview of the whole system. Further details are available in [2] and on the Dynamo project website (www.hpl.hp.com/cambridge/projects/Dynamo).

2. Overview

From a user's perspective, Dynamo looks like a PA-8000 software interpreter that itself runs on a PA-8000 processor (the hardware interpreter). Interpretation allows Dynamo to observe execution behavior without having to instrument the application binary. Since software interpretation is much slower than direct execution on the processor, Dynamo only interprets the instruction stream until a "hot" instruction sequence (or *trace*) is identified. At that point, Dynamo generates an optimized version of the trace (called a *fragment*) into a software code cache (called the *fragment cache*). Subsequent encounters of the hot trace's entry address during interpretation will cause control to jump to the top of the corresponding cached fragment. This effectively suspends the interpreter and allows the cached code to execute directly on the processor without incurring any further interpretive overhead. When control eventually exits the fragment cache, Dynamo resumes interpreting the instruction stream, and the process repeats itself.

Figure 1 illustrates this flow of control in more detail. Dynamo starts out by interpreting the input native instruction stream until a taken branch is encountered (A). If the branch target address corresponds to the entry point of a fragment already in the fragment cache (B), control jumps to the top of that fragment, effectively suspending Dynamo, and causing execution of the cached fragments to occur directly on the underlying processor (F). Otherwise, if the branch target satisfies a "start-of-trace" condition (C), a counter associated with the target address is incremented (D).

Our current prototype defines start-of-trace as targets of backward-taken branches (likely loop headers) and fragment cache exit branches (exits from previously identified hot traces). If the counter value exceeds a preset hot threshold (E), the interpreter toggles state and goes into "code generation mode" (G). When interpreting in this mode, the native instruction sequence being interpreted is recorded in a hot trace buffer, until an "end-of-trace" condition is reached (H). At that point the hot trace buffer is processed by a fast, lightweight optimizer (I) to create an optimized single-entry, multi-exit, contiguous sequence of instructions called the *fragment*[1]. Our current prototype defines end-of-trace as backward taken branches or taken branches whose targets correspond to fragment entry points in the fragment cache (i.e., fragment cache hits). A trace may also be truncated if its length exceeds a certain number of instructions. The fragment generated by the optimizer is emitted into the fragment cache by a linker (J), which also connects fragment exit branches to other fragments in the fragment cache if possible. Connecting fragments together in this manner minimizes expensive fragment cache exits to the Dynamo interpretive loop. The new fragment is tagged with the application binary address of the start-of-trace instruction.

As execution proceeds, the application's working set gradually materializes in the fragment cache, and the Dynamo *overhead* (time spent in the Dynamo interpretive loop / time spent executing in the fragment cache) begins to drop. Assuming that the majority of an application's execution time is typically spent in a small portion of its code, the performance benefits from repeated reuse of the optimized fragments can be sufficient to offset the overhead of Dynamo's operation. On the SpecInt95 benchmarks,

---

[1] A fragment is similar to a superblock, except for the fact that it is a dynamic instruction sequence, and can cross static program boundaries like procedure calls and returns.
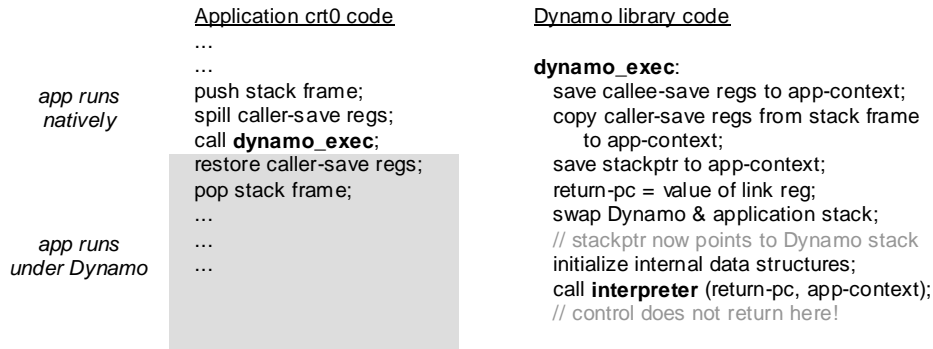
```
                    Application crt0 code          Dynamo library code
                    ...
                    ...                            dynamo_exec:
   app runs         push stack frame;                save callee-save regs to app-context;
   natively         spill caller-save regs;          copy caller-save regs from stack frame
                    call dynamo_exec;                   to app-context;
                    restore caller-save regs;        save stackptr to app-context;
                    pop stack frame;                 return-pc = value of link reg;
                    ...                              swap Dynamo & application stack;
   app runs         ...                              // stackptr now points to Dynamo stack
   under Dynamo     ...                              initialize internal data structures;
                                                     call interpreter (return-pc, app-context);
                                                     // control does not return here!
```

Figure 2. How Dynamo gains control over the application

the average Dynamo overhead is less than 1.5% of execution time. Dynamo's interpreter-based hot trace selection process (`A-H`) dominates this overhead, with the optimizer and linker components (`I, J`) contributing a relatively insignificant amount.

3. Startup and Initialization

Dynamo is provided as a user-mode dynamically linked library (shared library). The entry point into this library is the routine dynamo_exec. When dynamo_exec is invoked by an application, the remainder of the application code after return from the dynamo_exec call will execute under Dynamo control.

As outlined in Figure 2, dynamo_exec first saves a snapshot of the application's context (i.e., the machine registers and stack environment) to an internal app-context data structure. It then swaps the stack environment so that Dynamo's own code uses a custom runtime stack allocated separately for its use. Dynamo's operation thus does not interfere with the runtime stack of the application running on it. The interpreter (box `A` in Figure 1) is eventually invoked with the return-pc corresponding to the application's dynamo_exec call. The interpreter starts interpreting the application code from this return-pc, using the context saved in app-context. The interpreter never returns to dynamo_exec (unless a special bailout condition occurs, which is discussed later), and Dynamo has gained control over the application. From this point onwards, an application instruction is either interpreted, or a copy of it is executed in the fragment cache. The original instruction is never executed in place the way it would have been if the application were running directly on the processor.

We provide a custom version of the execution startup code crt0.o, that checks to see if the Dynamo library is installed on the system, and if it is, invokes dynamo_start prior to the jump to _start (the application's main entry point). Application binaries that are linked with this version of crt0.o will transparently invoke Dynamo if Dynamo is installed on the system, otherwise they will execute normally. The application binary itself remains unchanged whether or not it is run under Dynamo. This strategy allows Dynamo to preserve the original mapping of the application's text segment, a key requirement for transparent operation.

As part of the initialization done in dynamo_exec prior to actually invoking the interpreter, Dynamo *mmaps* a separate area of memory that it manages itself. All dynamically allocated objects in Dynamo code are created in this area of memory. Access to this area is protected to prevent the application from inadvertently or maliciously corrupting Dynamo's state.

4. Fragment Formation

Due to the significant overheads of operating at runtime, Dynamo has to maximize the impact of any optimization that it performs. Furthermore, since the objective is to complement, not compete, with the compiler that generated the instruction stream, Dynamo primarily looks for performance opportunities that tend to manifest themselves in the runtime context of the application. These are generally redundancies that cross static program boundaries like procedure calls, returns, virtual function calls, indirect branches and dynamically linked function calls. Another performance opportunity is instruction cache utilization, since a dynamically contiguous sequence of frequently executing instructions may often be statically non-contiguous in the application binary.

Dynamo's unit of runtime optimization is a *trace*, defined as a dynamic sequence of consecutively executed instructions. A trace starts at an address that satisfies the start-of-trace condition and ends at an address that satisfies the end-of-trace condition. Traces may extend across statically or dynamically linked procedure calls/returns, indirect branches and virtual function calls. Dynamo first selects a "hot" trace, then optimizes it, and finally emits relocatable code for it into the fragment cache. The emitted relocatable code is contiguous in the fragment cache memory, and branches that exit this code jump to corresponding exit stubs at the bottom of the code. This code is referred to as a *fragment*. The trace is a unit of the application's dynamic instruction stream (i.e., a sequence of application instructions whose addresses are application binary addresses) whereas the fragment is a Dynamo internal unit, addressed by fragment cache addresses. The following subsections outline the trace selection, trace optimization and fragment code generation mechanisms of Dynamo.

4.1 Trace selection

Since Dynamo operates at runtime, it cannot afford to use elaborate profiling mechanisms to identify hot traces (such as [14][4]). Moreover, most profiling techniques in use today have been designed for offline use, where the gathered profile data is collated and analyzed post-mortem. The objective here is not accuracy, but predictability. If a particular trace is very hot over a short period of time, but its overall contribution to the execution time is small, it may still be an important trace to identify. Another concern for Dynamo is the amount of counter updates and counter storage required for identifying hot traces, since this adds to the overhead and memory footprint of the system.

As discussed in Section 2, Dynamo uses software interpretation of the instruction stream to observe runtime execution behavior. Interpretation is expensive but it prevents the

need to instrument the application binary or otherwise perturb it in any way. Interpretation is preferable to statistical PC sampling because it does not interfere with applications that use timer interrupts. Also, as we will elaborate shortly, interpretation allows Dynamo to select hot regions directly without having to collate and analyze point statistics like the kind produced by PC sampling techniques. Another important advantage of interpretation is that it is a deterministic trace selection scheme, which makes the task of engineering the Dynamo system much easier.

It is worth noting that the "interpreter" here is a native instruction interpreter and that the underlying CPU is itself a very fast native instruction interpreter implemented in hardware. This fact can be exploited on machines that provide fast breakpoint traps (e.g., through user-mode accessible breakpoint window registers) to implement the Dynamo interpreter very efficiently [2]. On the PA-8000 however, breakpoint traps are very expensive, and it was more efficient to implement the interpreter by using emulation. The higher the interpretive overhead, the earlier Dynamo has to predict the hot trace in order to keep the overheads low. In general, the more speculative the trace prediction scheme, the larger we need to size the fragment cache, to compensate for the larger number of traces picked as a result. Thus, the interpretive overhead has a ripple effect throughout the rest of the Dynamo system.

Dynamo uses a speculative scheme we refer to as MRET (for *most recently executed tail*) to pick hot traces without doing any path or branch profiling. The MRET strategy works as follows. Dynamo associates a counter with certain selected start-of-trace points such as the target addresses of backward taken branches. The target of a backward taken branch is very likely to be a loop header, and thus the head of several hot traces in the loop body. If the counter associated with a certain start-of-trace address exceeds a preset threshold value, Dynamo switches its interpreter to a mode where the sequence of interpreted instructions is recorded as they are being interpreted. Eventually, when an end-of-trace condition is reached, the recorded sequence of instructions (the most recently executed tail starting from the hot start-of-trace) is selected as a hot trace.

The insight behind MRET is that when an instruction becomes hot, it is statistically likely that the very next sequence of executed instructions that follow it is also hot. Thus, instead of profiling the branches in the rest of the sequence, we simply record the tail of instructions following the hot start-of-trace and optimistically pick this sequence as a hot trace. Besides its simplicity and ease of engineering, MRET has the advantage of requiring much smaller counter storage than traditional branch or path profiling techniques. Counters are only maintained for potential loop headers. Furthermore, once a hot trace has been selected and emitted into the fragment cache, the counter associated with its start-of-trace address can be recycled. This is possible because all future occurrences of this address will cause the cached version of the code to be executed and no further profiling is required.

Subsequent hot traces that also start at the same start-of-trace address will be selected when control exits the first selected trace for that start-of-trace address. Exits from previously selected hot traces are treated as start-of-trace points by Dynamo (see Figure 1). This allows subsequent hot tails that follow the earlier hot start-of-trace to be selected by the MRET scheme in the usual manner.

No profiling is done on the code generated into Dynamo's fragment cache. This allows the cached code to run directly on the processor at full native speed without any Dynamo introduced overheads. The flip side of this is that if the biases of some branches change after a hot trace was selected, Dynamo would be unable to detect it. In order to allow Dynamo to adapt to changing branch biases, the fragment cache is designed to tolerate periodic flushes. Periodically flushing some of the traces in the fragment cache helps remove unused traces, and also forces re-selection of active traces. This is discussed in more detail in Section 6.

4.2 Trace optimization

The selected hot trace is prepared for optimization by converting it into a low-level intermediate representation (IR) that is very close to the underlying machine instruction set.

The first task of trace optimization is to transform the branches on the trace so that their fall-through direction remains on the trace. Loops are only allowed if the loop-back branch targets the start-of-trace. Otherwise the loop-back branch is treated as a trace exit. Unconditional direct branches are redundant on the trace
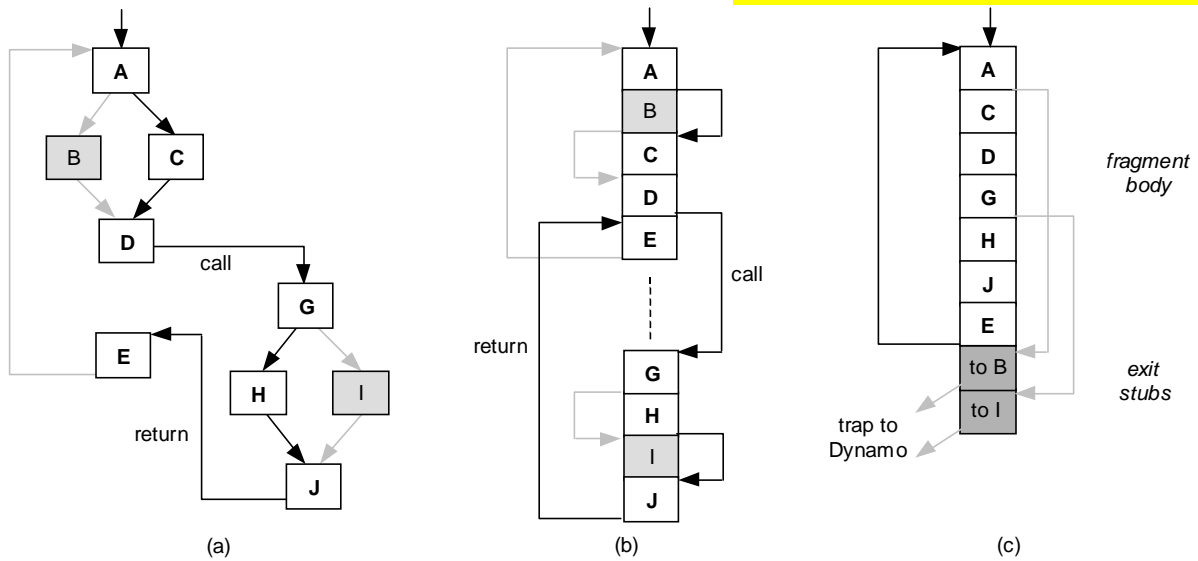

Figure 3. Control flow snippet in the application binary, (b) Layout of this snippet in the application program's memory, and (c) Layout of a trace through this snippet in Dynamo's fragment cache.

==and can be removed.== In the case of branches with side-effects, such as branch-and-link branches, the side-effect is preserved even if the branch itself is removed. After trace optimization, no branch-and-link type branches remain on the trace.

Even indirect branches may be redundant. For example, a return branch if preceded by the corresponding call on the trace is redundant and will be removed. ==Other indirect branches are optimistically transformed into direct conditional branches.== The transformed conditional branch compares the dynamic branch target with the target contained in the trace at the time the trace was selected (referred to as the *predicted* indirect branch target). If the comparison succeeds, control goes to the predicted (on-trace) target. If the comparison fails, control is directed to a special Dynamo routine that looks up a Dynamo-maintained switch table. The switch table is a hash table indexed by indirect branch target addresses (application binary addresses). The table entries contain the fragment cache address corresponding to the target. If an entry is found for the dynamic indirect branch target, control is directed to the corresponding fragment cache address. Otherwise, control exits the fragment cache to the Dynamo interpreter. If the interpreter then selects a new hot trace starting at that dynamic indirect branch target, Dynamo will add a new entry to the switch table corresponding to the mapping from the start-of-trace application address to its fragment cache address. Assuming execution follows the selected hot trace most of the time, this transformation replaces a potentially expensive indirect branch with a less expensive direct conditional branch. The following outlines the transformed code for an indirect branch instruction:

```
// assuming the indirect branch's dynamic target is in Rx
spill Rscratch to app-context; // free a fixed register
set Rscratch = address of predicted on-trace target;
if (Rx = = Rscratch) goto predicted target;
copy Rx to Rscratch;
goto switch_table_lookup(Rscratch);
```

The actual register that contains the original indirect branch's dynamic target can be different for different indirect branch instructions. The purpose of copying this dynamic target to register Rscratch is to ensure that when control enters the switch table lookup routine at execution time, the same fixed register (Rscratch) will contain the dynamic target that has to be looked up.

Finally, an unconditional trace exit branch is appended to the bottom of the trace so that control reaching the end of the trace can exit it via a taken branch. After fixing up the branches on the trace, the result is a single-entry, multi-exit sequence of instructions with no internal control join points. Figure 3 illustrates the branch adjustments that occur after a trace is selected from the application binary.

Since traces are free of internal join points, new opportunities for optimization may be exposed that were otherwise unsafe in the original program code. The simplicity of control flow allowed within a trace also means traces can be analyzed and optimized very rapidly. In fact, the Dynamo trace optimizer is non-iterative, and optimizes a trace in only two passes: a forward pass and a backward pass. During each pass the necessary data flow information is collected as it proceeds along the fragment. Most of the optimizations performed involve redundancy removal: redundant branch elimination, redundant load removal, and redundant assignment elimination. These opportunities typically result from partial redundancies in the original application binary that become full redundancies in a join-free trace.

The trace optimizer also sinks all partially redundant instructions (i.e., on-trace redundancies) into special off-trace *compensation blocks* that it creates at the bottom of the trace. This ensures that the partially redundant instructions get executed only when control exits the trace along a specific path where the registers defined by those instructions are downward-exposed. Fragment A in Figure 5 illustrates such a case. The assignment to register r5 shown in the compensation block (thick border) could have originally been in the first trace block. This sinking code motion ensures that the overhead of executing this assignment is only incurred when control exits the fragment via the path along which that assignment to r5 is downwards exposed.

==Other conventional optimizations performed are copy propagation, constant propagation, strength reduction, loop invariant code motion and loop unrolling.== Dynamo also performs runtime disambiguated conditional load removal by inserting instruction guards that conditionally nullify a potentially redundant load.

Note that load removal is only safe if it is known that the respective memory location is not volatile. Information about volatile variables may be communicated to Dynamo through the symbol table. In the absence of any information about volatile variables, load removal transformations are conservatively suppressed.

4.3 Fragment code generation

The fragment code generator emits code for the trace IR into the fragment cache. The emitted code is referred to as a *fragment*. The fragment cache manager (discussed in Section 6) first allocates sufficient room in the fragment cache to generate the code.

A trace IR may be split into multiple fragments when it is emitted into the fragment cache. This is the case, for example, if a direct conditional branch is encountered on the trace, which was converted from the application's original indirect branch instruction by the trace optimizer (see Section 4.2). Such a branch splits the trace into two fragments. The predicted on-trace target of the original indirect branch, which is the instruction immediately following this branch on the trace, starts a separate fragment.

Virtual registers may be used in the IR but the trace optimizer retains their original machine register mappings. The register allocator attempts to preserve the original machine register mappings to the extent possible when the code is finally emitted. The allocator reservers one register to hold the address of the app-context data structure (see Figure 2) when control is within the fragment. The app-context is a Dynamo internal data structure that is used to keep the application's machine state during interpretation, and also to record a snapshot of the application's machine state at the point of the last fragment cache exit to Dynamo. The trace optimizer uses the app-context as a spill area to create temporary scratch registers necessary for its optimizations. It cannot use the application's runtime stack as a spill area because that would interfere with stack operations generated by the static compiler that created the application binary.

==Generation of the fragment code from the trace IR involves two steps: emitting the fragment body, and emitting the fragment exit stubs.== Emitting the fragment body involves straightforward generation of the code corresponding to the trace IR itself. After that, a unique exit stub is emitted for every fragment exit branch and fragment loop-back branch. The exit stub is a piece of code that transfers control from the fragment cache to the Dynamo interpreter in a canonical way, as outlined below:
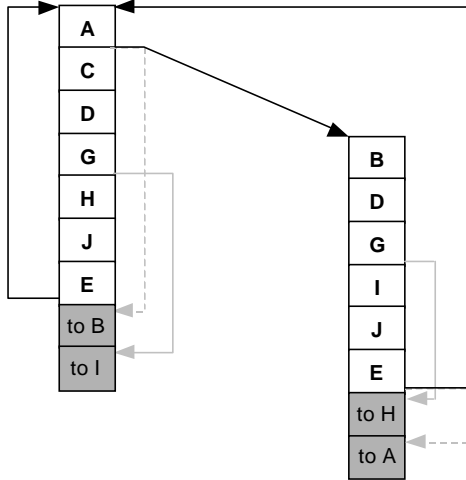
Figure 4. Example of fragment linking

Figure 5. Example of link-time optimization

```
spill Rlink to app-context;
branch & link to interpreter; // sets Rlink to the following PC
<ptr to linkage info for this exit branch>
```

Each stub can be entered by only one fragment exit branch. The stub code first saves the link register (Rlink) to the app-context. It then does a branch and link to the entry point of the Dynamo interpreter, which sets the Rlink register to the fragment cache address following this branch. The Dynamo interpreter will take a snapshot of the application's machine state (with the application's original Rlink value being taken from the app-context data structure) prior to starting interpretation. The end of the exit stub beyond the branch and link instruction contains a pointer to linkage information for the fragment exit branch associated with the stub. When control exits the fragment to the Dynamo interpreter, the interpreter consults this linkage information to figure out the next application address at which it should start interpretation. The value of the Rlink register contains the address of the location containing the pointer to the linkage information for the current fragment exit.

5. Fragment Linking

After the fragment code is emitted into the fragment cache, the new fragment is linked to other fragments already in the fragment cache. Linking involves patching a fragment exit branch so that its taken target is the entry point of another fragment, instead of to its exit stub.

As an example, suppose the trace BDGIJE in Figure 3 (a) now becomes hot (B is a valid start-of-trace by our definition, when it is entered via an exit from the earlier hot trace ACDGHJE). Figure 4 illustrates the linking that occurs after the fragment corresponding to the BDGIJE trace is emitted into the fragment cache. Linked branches are shown as dark arrows, and their original unlinked versions are indicated as dashed light arrows.

Fragment linking is essential for performance, because it prevents expensive exits from the fragment cache back to the Dynamo interpreter. In our prototype implementation on the PA-8000 for example, disabling fragment linking results in an order of magnitude slowdown (by an average factor of 40 for the SpecInt95 benchmarks).

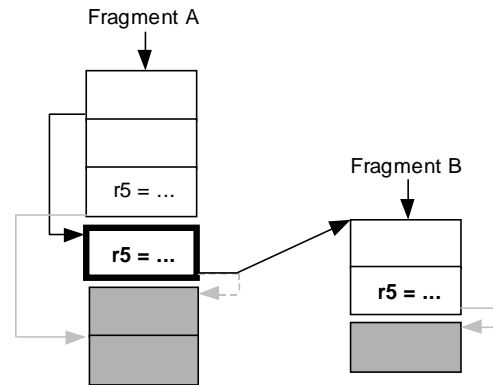Fragment linking also provides an opportunity for removing redundant compensation code from the source fragment involved in the link. Recall that the trace optimizer sinks on-trace redundancies into compensation blocks, so that these instructions are only executed when control exits the fragment along a particular path (see Section 4.2). Fragment A in Figure 5 illustrates such a case, where the assignment to r5 shown in the compensation block (thick border) was originally in the first block before it was sunk into its compensation block. As part of the linkage information that is kept at each fragment exit stub (the shaded boxes in Figure 5), a mask of on-trace redundant register assignments along that particular fragment exit is maintained. In Figure 5, this mask would be kept in the exit stub corresponding to the compensation block, and bit 5 of the mask would be set. A similar mask of killed register assignments at the top of every fragment is also maintained as part of the Dynamo internal data structure that keeps fragment-related information. At link-time, if a register appears in both masks, the instruction that last defined it in the source fragment's compensation block is dead and can be removed. This is illustrated in Figure 5, where the assignment to r5 in Fragment A's compensation block can be deleted because r5 is defined before being used on entry to Fragment B.

While the advantages of linking are clear, it also has some disadvantages that impact other parts of the Dynamo system. For instance, linking makes the removal of individual fragments from the fragment cache expensive, because all incoming branches into a fragment must first be unlinked first. Linking also makes it difficult to relocate fragments in the fragment cache memory after they have been emitted. This might be useful for instance to do periodic de-fragmentation of the fragment cache memory.

6. Fragment Cache Management

Dynamo cannot afford to do complicated management of the fragment cache storage, because of the overheads this would incur. We could avoid storage management altogether by simply expanding the size of the fragment cache as needed. But this has several undesirable effects. For example, one of the advantages of collecting hot traces in a separate fragment cache is the improved instruction cache locality and TLB utilization that can result from keeping the working set close together in memory. This advantage could go away if over time, the hot traces that make up the current working set are spread out over a large area of fragment cache memory. Clearly, the ideal situation where the fragment cache only contains the traces that make up the current working set is difficult
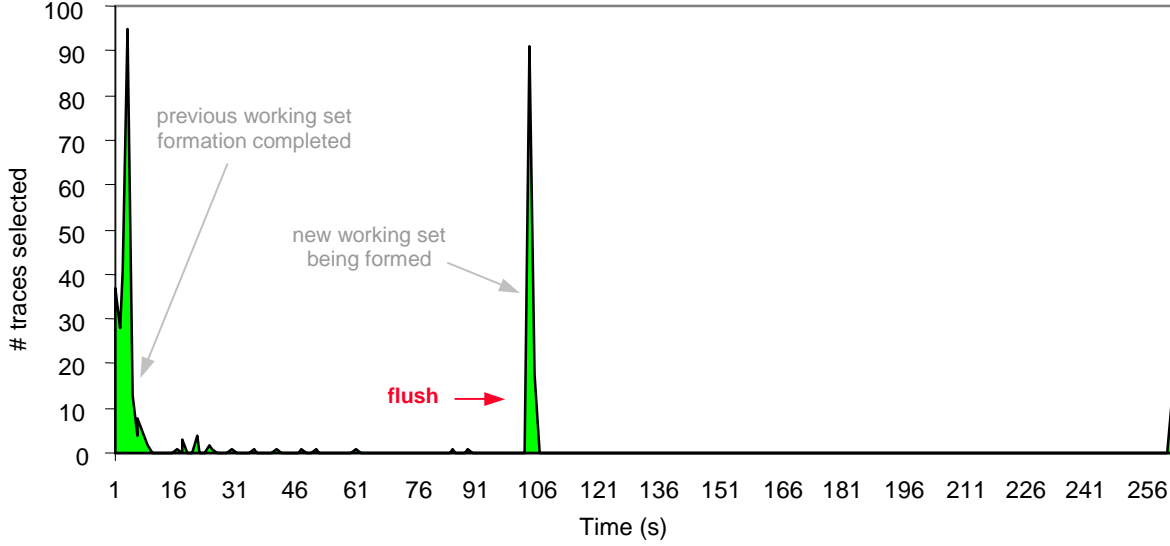
Figure 6. Dynamic trace selection rate for m88ksim, showing a sharp change in the working set ~106 sec into its execution

to achieve. The overhead of implementing an LRU type scheme to identify cold fragments would be too expensive as well. Moreover, as pointed out earlier, any policy that only removes a few fragments would incur the expense of having to unlink every incoming branch into these fragments.

Dynamo instead employs a novel pre-emptive flushing heuristic to periodically remove cold traces from the fragment cache without incurring a high penalty. A complete fragment cache flush is triggered whenever Dynamo recognizes a sharp increase in the fragment creation rate (or hot trace selection rate). The rationale here is that a sharp rise in new fragment creation is very likely indicative of a significant change in the working set of the program that is currently in the fragment cache. Since control is predominantly being spent in Dynamo during this stage, the fragment cache flush is essentially "free". Figure 6 illustrates this scenario for the SpecInt95 m88ksim benchmark. Since all fragments are removed during a fragment cache flush, no unlinking of branches needs to be done.

The pre-emptive flushing mechanism has other useful side effects. All fragment-related data structures maintained for internal bookkeeping by Dynamo are tied to the flush, causing these memory pools to be reset as a side effect of a pre-emptive flush. A pre-emptive flush thus serves as an efficient garbage collection mechanism to free dynamic objects associated with fragments that are likely to have dropped out of the current working set. If some fragments belonging to the new working set are inadvertently flushed as a result, they will be regenerated by Dynamo when those program addresses are encountered later during execution. Regeneration of fragments allows Dynamo to adapt to changes in the application's branch biases. When a trace is re-created, Dynamo may select a different tail of instructions from the same start-of-trace point. This automatic "re-biasing" of fragments is another useful side effect of the pre-emptive cache flushing strategy.

7. Signal Handling

Optimizations that involve code reordering or removal, such as dead code elimination and loop unrolling, can create a problem if a signal arrives while executing the optimized fragment, by making it difficult or impossible for Dynamo to recreate the original signal context prior to the optimization. This can create complications for precise signal delivery. For example, the application might arm a signal with a handler that examines or even modifies the machine context at the instant of the signal. If a signal arrives at a point where a dead register assignment has been removed, the signal context is incomplete.

Dynamo intercepts all signals, and executes the program's signal handler code under its control, in the same manner that it executes the rest of the application code (box K in Figure 1). This gives Dynamo an opportunity to rectify the signal context that would otherwise be passed directly to the application's handler by the operating system. Asynchronous signals (such as keyboard interrupts, etc., where the signal address is irrelevant) are treated differently from synchronous signals (such as segment faults, etc., where the signal address is critical).

If an asynchronous signal arrives when executing a fragment, the Dynamo signal handler will queue it and return control back to the fragment cache. All queued asynchronous signals are processed when the next normal fragment cache exit occurs. This allows Dynamo to provide a proper signal context to the application's handler since control is not in the middle of an optimized fragment at the time the signal context is constructed.

In order to bound asynchronous signal handling latency, the Dynamo signal handler unlinks all linked branches on the current fragment prior to resuming execution of the fragment. To disconnect self-loops in a similar manner, the fragment generator emits an exit stub for each self-loop branch in addition to the exit stubs for the fragment exit branches. Unlinking the current fragment forces the next fragment exit branch to exit the fragment cache via the exit stub, preventing the possibility of control spinning within the fragment cache for an arbitrarily long period of time before the queued signals are processed. This feature allows Dynamo to operate in environments where soft real-time constraints must be met.

Synchronous signals on the other hand are problematic, because they cannot be postponed. A drastic solution is to suppress code removing and reordering transformations altogether. A more acceptable alternative is to use techniques similar to that developed for debugging of optimized code to de-optimize the fragment code before attempting to construct the synchronous signal context.
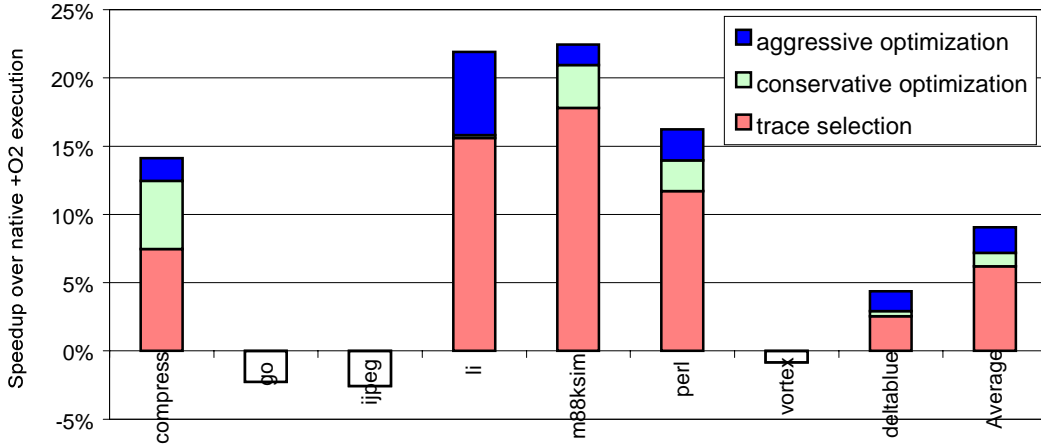
7

Figure 7. Speedup of +O2 optimized PA-8000 binaries running on Dynamo, relative to the identical binaries running standalone. The contributions from dynamic inlining due to trace selection, conservative trace optimization and aggressive trace optimization are shown. Dynamo bails out to direct native execution on *go* and *vortex*.

Fortunately, the problem of de-optimizing is much simpler in Dynamo since only straight-line fragments are considered during optimization. Optimization logs can be stored along with each fragment that describes compensation actions to be performed upon signal-delivery, such as the execution a previously deleted instruction. This is presently an ongoing effort in the Dynamo project.

Our prototype currently implements a less ambitious solution to this problem, by dividing trace optimizations into two categories, *conservative* and *aggressive*. Conservative optimizations allow the precise signal context to be constructed if a synchronous fault occurs while executing the fragment. Aggressive optimizations on the other hand cannot guarantee this. Examples of conservative optimizations include constant propagation, constant folding, strength reduction, copy propagation and redundant branch removal. The aggressive category includes all of the conservative optimizations plus dead code removal, code sinking and loop invariant code motion. Certain aggressive optimizations, like redundant load removal, can sometimes be incorrect, if the load is from a volatile memory location.

Dynamo's trace optimizer is capable of starting out in its aggressive mode of optimization, and switching to conservative mode followed by a fragment cache flush if any suspicious instruction sequence is encountered. Unfortunately, the PA-RISC binary does not provide information about volatile memory operations or information about program-installed signal handlers. So this capability is currently unused in Dynamo. In a future version of Dynamo, we plan to investigate ways to allow the generator of Dynamo's input native instruction stream to provide hints to Dynamo. Dynamo can use such hints if they are available, but will not rely on them for operation.

8. Performance Data

For performance evaluation we present experiments on several integer benchmarks. Dynamo incurs a fixed startup overhead for allocating and initializing its internal data structures and the fragment cache. The startup overhead could probably be improved through more careful engineering. But for the purposes of this study, we use benchmarks that long enough to allow the startup and initialization overhead to be recouped. This section

presents data comparing the performance of running several integer benchmarks on Dynamo to the identical binary executing directly on the processor. Our benchmark set includes the SpecInt95 benchmarks[2] and a commercial C++ code called *deltablue*, which is an incremental constraint solver [28]. The programs were compiled at the +O2 optimization level (equivalent to the default –O option) using the product HP C/C++ compiler. This optimization level includes global intraprocedural optimization. Performance measurements were based on wall clock time on a lightly loaded single-processor HP PA-8000 workstation [21] running the HP-UX 10.20 operating system.

Figure 7 shows the speedup that Dynamo achieves over +O2 optimized native program binaries running without Dynamo. For these runs, Dynamo was configured to use a fixed size 150 Kbyte fragment cache, which is flushed when sharp changes occur to the trace selection rate or there is no room to generate new fragments. Details about the performance impact of varying the fragment cache size are outside the scope of this paper and can be found elsewhere [2]. As the figure indicates, Dynamo achieves considerable speedup in some cases, over 22% in *li* and *m88skim*, about 18% in *perl,* and about 14% in *compress*. These four programs have relatively stable working sets, a fact that dynamic optimization can exploit very well. The average overall speedup is about 9%. A significant portion of the performance gains come from the act of selecting a trace and forming a fragment out of it, that is, from the implied partial procedure inlining and improved

---

[2] Our experiments do not include the SpecInt95 *gcc* benchmark. This benchmark actually consists of repeated runs of *gcc* on a number of input files, and the individual runs are too short running to qualify for our performance study (less than 60 seconds on the PA-8000). To understand the performance characteristics of *gcc*, we modified the *gcc* program to internally loop over the input files, thus resulting in a single long invocation of *gcc*. We do not show data for the modified *gcc* because it does not represent the original benchmark, but it's performance characteristics are comparable to that of *go* for all of the data shown here.
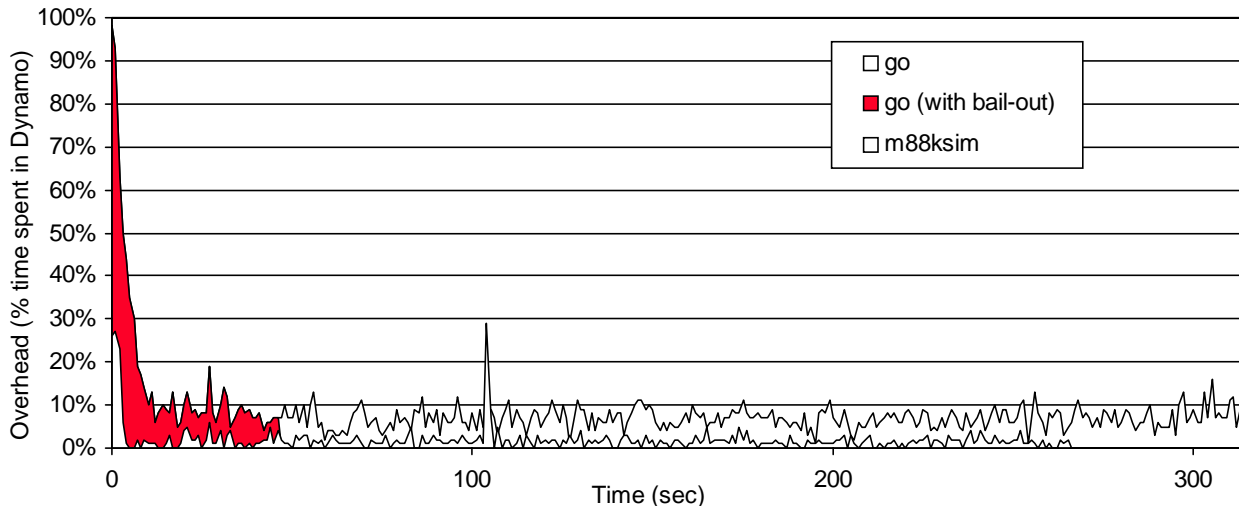
Figure 8. Illustration of bail-out. Dynamo bails out on *go* ~45 sec into its execution, after which *go* runs directly on the processor without incurring any Dynamo overhead. *m88ksim* is shown for comparison as a case where Dynamo does not bail out.

code layout in the fragment cache. Fragment optimization accounts for approximately 3% of the total gains on average, and one-third of this is due to conservative (signal and volatile-memory safe) optimizations. Note however, that if we ignore the inputs on which Dynamo bails out (as discussed shortly), the average contribution due to trace optimization is around 5%.

Dynamo does not achieve performance improvements on programs *go*, *ijpeg* and *vortex*. Dynamo's startup time is a non-negligible fraction of the total runtime of *ijpeg*, as *ijpeg* does not run long enough to recoup Dynamo's startup overhead before starting to provide any performance benefit. In the case of *go* and *vortex* that run for a long time, the problem is the lack of a stable working set. A relatively high number of distinct dynamic execution paths are executed in these benchmarks [4]. Frequently changing dynamic execution paths result in an unstable working set, and Dynamo spends too much time selecting traces without these traces being reused sufficiently in the cache to offset the overhead of its own operation.

Fortunately, since Dynamo is a native-to-native optimizer, it can use the original input program binary as a fallback when its overhead starts to get too high. Dynamo constantly monitors the ratio of time spent in Dynamo over time spent in the fragment cache. If this ratio stays above a tolerable threshold for a prolonged period of time, Dynamo assumes that the application cannot be profitably optimized at runtime. At that point Dynamo *bails-out* by loading the application's app-context to the machine registers and jumping to an application binary address. From that point on the application runs directly on the processor, without any further dynamic optimization. Bail-out allows Dynamo to come close to break-even performance even on "ill-behaved" programs with unstable working sets. This is illustrated in the graph in Figure 8 for the benchmark *go*. The Dynamo overhead for a relatively well-behaved application, *m88ksim*, is also shown for comparison.

Figure 9 shows Dynamo's performance on binaries compiled with higher optimization levels. The figure shows the program runtimes with and without Dynamo, for three optimization levels: +O2 (same as –O), +O4, and profile-based +O4 +P (i.e., +O4 with a prior profile collection run). At level +O4, the HP C compiler performs global interprocedural and link-time optimization. At level +O4 +P the compiler performs +O4 optimizations based on profile information gathered during a prior +O4 run. However, compile-time increases very significantly from +O2 to +O4, and the ability to debug the binary is lost. Because of this, most software vendors are reluctant to enable higher optimization levels, in spite of the performance advantages they offer.

The data in Figure 9 shows that Dynamo finds performance improvement opportunities even in highly optimized binaries. In fact, on this set of benchmarks, Dynamo is able to raise the average performance of +O2 compiled binaries to a level that slightly exceeds the performance of their +O4 compiled versions running without Dynamo! This performance boost comes in a transparent fashion, without the creator of the binary having to do anything special. The fact that Dynamo finds performance improvement opportunities even in +O4 optimized binaries is not as surprising as it first seems, because Dynamo primarily focuses on runtime performance opportunities that a static compiler would find difficult to exploit.

In some programs (such as *li* and *perl*), Dynamo is able to boost the performance of even profile-feedback compiled binaries (+O4 +P). On average however, the benefits of Dynamo disappear once static optimization is enhanced with profile information. This is to be expected, as the most beneficial inlining and other path-sensitive optimizations have been already made at compile-time.

As pointed out in the introduction, the goal of this study is to establish the limits of Dynamo's capabilities in an extreme setting, where the quality of the input program code is good. In compiling these benchmarks, the static compiler had all of the program sources available, and no dynamically linked libraries were used. Using good quality compiled code as input forced the development
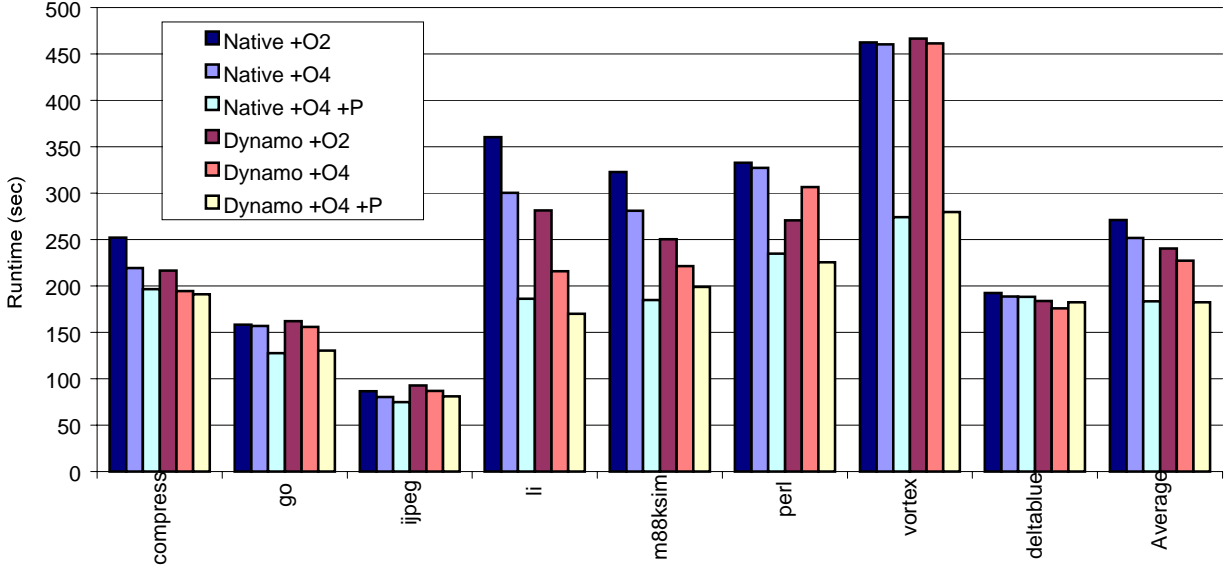
Figure 8. Dynamo performance on native binaries compiled at higher optimization levels (the first 3 bars for each program correspond to the native runs without Dynamo, and the next 3 bars correspond to the runs on Dynamo)

effort to focus on fine-tuning the engineering of the Dynamo system.

It should be emphasized that the performance data shown here is very specific to the quality of the code produced by the PA-8000 compiler, and to the PA-8000 processor implementation. Although the hot trace selection and dynamic optimization can be expected to provide benefits in general, the actual impact in terms of wall-clock performance improvement will vary from target to target. On the deeply pipelined PA-8000 for example, the branch misprediction penalty is 5 cycles, and indirect branches (including returns) are always mispredicted. Indirect branch removal therefore makes a big contribution toward Dynamo's performance gains on the PA-8000. On the other hand, the PA-8000 has a large instruction cache (1 Mbyte), so the gains from improved I-cache locality in the software fragment cache code are unlikely to be significant. However, the processor has a unified instruction and data TLB with only 96 entries, so the reduction in TLB pressure due to better locality of the working set in the fragment cache can contribute to a performance boost.

9. Related Work

In focusing on native-to-native runtime optimization, Dynamo is a fundamentally different approach from past work on dynamic compilation. Just-in-time compilers delay all compilation until runtime [6][11][10]. Selective dynamic compilation [1][9][23][13][22][26][16][24] is a staged form of compilation that restricts dynamic compilation to selected portions of code identified by user annotations or source language extensions. In these cases, the static compiler prepares the dynamic compilation process as much as possible by generating templates that are instantiated at run-time by a specialized dynamic compiler.

In contrast to both just-in-time and selective dynamic compilation, Dynamo separates that task of compilation, which occurs prior to execution, from dynamic optimization, which occurs entirely at runtime and without requiring user assistance. Dynamo's input is an already compiled native instruction stream, that is re-optimized to exploit performance opportunities that manifest themselves at runtime.

A lot of work has been done on dynamic translation as a technique for non-native system emulation [8][30][5][31][12][17]. The idea is to lower emulation overhead by caching native code translations of frequently interpreted regions. Unlike such binary translators, Dynamo is not concerned with translation. The Dynamo approach does however allow one to couple a fast lightweight translator that emits native code to Dynamo, which then becomes a backend optimizer.

There are several implementations of *offline binary translators* that also perform native code optimization [7][29]. These generate profile data during the initial run via emulation, and perform background translation together with optimization of hot spots based on the profile data. The benefit of the profile-based optimization is only available during subsequent runs of the program and the initial profile-collecting run may suffer from worsened performance.

Hardware solutions for a limited form of runtime code optimization are now commonplace in modern superscalar microprocessors [21][25][19]. The optimization unit is a fixed size instruction window, with the optimization logic operating on the critical execution path. The Trace Cache is another hardware alternative that can be extended to do superscalar-like optimization off the critical path [27][15]. Dynamo offers the potential for a purely software alternative, which could allow it to be tailored to specific application domains, and cooperate with the compiler or JIT in ways that hardware dynamic optimizers cannot.

10. Conclusion

Dynamo is a novel performance delivery mechanism. It complements the compiler's traditional strength as a static performance improvement tool by providing a dynamic optimization capability. In contrast to other approaches to dynamic optimization, Dynamo works transparently, requiring no user intervention. This fact allows Dynamo to be bundled with a computer system, and shipped as a client-side performance delivery mechanism, whose activation does not depend on the ISVs (independent software vendors) in the way that traditional compiler optimizations do.

This paper demonstrates that it is possible to engineer a practical software dynamic optimizer that provides a significant performance benefit even on highly optimized executables produced by a static compiler. The key is to focus the optimization effort on opportunities that are likely to manifest themselves only at runtime, and hence those that a static compiler might miss.

We are currently investigating applications of Dynamo's dynamic optimization technology in many different areas. One of the directions we are exploring is to export an API to the application program, so that a "Dynamo-aware" application can use the underlying system in interesting ways. This might be useful for example to implement a very low-overhead profiler, or a JIT compiler. From Dynamo's perspective, user and/or compiler hints provided via this API might allow it to perform more comprehensive optimizations that go beyond the scope of individual traces. Finally, we are also looking at the problem of transparent de-optimization at runtime.

11. Acknowledgements

12. References

[1] Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., and Bershad, B.N. 1996. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*.

[2] Bala, V., Duesterwald, E., and Banerjia, S. 1999. Transparent dynamic optimization: The design and implementation of Dynamo. *Hewlett Packard Laboratories Technical Report HPL-1999-78. June 1999.*

[3] Bala V., and Freudenberger, S. 1996. Dynamic optimization: the Dynamo project at HP Labs Cambridge (project proposal). HP Labs internal memo, Feb 1996.

[4] Ball, T., and Larus, J.R. 1996. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris*. 46-57.

[5] Bedichek, R. 1995. Talisman: fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

[6] Chambers, C., and Ungar, D. 1989. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*. 146-160.

[7] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, B., and Yates, J. 1998. FX!32: a profile-directed binary translator. *IEEE Micro, Vol 18, No. 2, March/April 1998.*

[8] Cmelik, R.F., and Keppel, D. 1993. Shade: a fast instruction set simulator for execution profiling. *Technical Report UWCSE-93-06-06, Dept. Computer Science and Engineering, University .of Washington.*

[9] Consel, C., and Noel, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages.* 145-156.

[10] Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. 1997. Compiling Java Just In Time. *IEEE Micro*, May/Jun 1997.

[11] Deutsch, L.P. and Schiffman A.M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*. 297-302.

[12] Ebcioglu K., and Altman, E.R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 26-37.

[13] Engler, D.R. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*.

[14] Fisher, J., and Freudenberger, S. 1992. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 5)*. Oct 1992. 85-95.

[15] Friendly, D.H., Patel, S.J., and Patt., Y.N. 1998. Putting the fill unit to work: dynamnic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual Internation Symposium on Microarchitecture (MICRO-31), Dallas*. 173-181.

[16] Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S.J. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*. 293-303.

[17] Herold, S.A. 1998. Using complete machine simulation to understand computer system behavior. *Ph.D. thesis, Dept. Computer Science, Stanford University.*

[18] Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P. P., Warter, N.J., Bringmann, R.A., Ouellette, R.Q., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. 1993.The superblock: an effective structure for VLIW and superscalar compilation. *The Journal of Supercomputing 7*, (Jan.). 229-248.

[19] Keller, J. 1996. The 21264: a superscalar Alpha processor with out-of-order execution. Presented at the 9th Annual Microprocessor Forum, San Jose, CA.

[20] Kelly, E.K., Cmelik, R.F., and Wing, M.J. 1998. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. *U.S. Patent 5,832,205, Nov. 1998.*

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

[22] Leone, M. and Dybvig, R.K. 1997. Dynamo: a staged compiler architecture for dynamic program optimization. *Technical Report #490, Dept. of Computer Science, Indiana University.*

[23] Leone, M. and Lee, P. 1996. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*. 137-148.

[24] Marlet, R., Consel, C., and Boinot, P. Efficient incremental run-time specialization for free. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*. 281-292.

[25] Papworth, D. 1996. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, (Apr.). 8-15.

[26] Poletta, M., Engler, D.R., and Kaashoek, M.F. 1997. tcc: a system for fast flexible, and high-level dynamic code generation. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*. 109-121.

[27] Rotenberg, E., Bennett, S., and Smith, J.E. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris*. 24-35.

[28] Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. 1993. Multi-way versus one-way constraints in user interfaces: experiences with the Deltablue algorithm. *Software – Practice and Experience 23*, 5 (May). 529-566.

[29] Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., and Robinson, S.G. Binary Translation. *Digital Technical Journal, Vol 4, No. 4, Special Issue, 1992.*

[30] Stears, P. 1994. Emulating the x86 and DOS/Windows in RISC environments. In *Proceedings of the Microprocessor Forum*, San Jose, CA.

[31] Witchel, E. and Rosenblum R. 1996. Embra: fast and flexible machine simulation. In *Proceedings of the SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*. 68-78