Selective Runtime Memory Disambiguation in a Dynamic Binary Translator

Bolei Guo[1] Youfeng Wu[2] Cheng Wang[2] Matthew J. Bridges[1] Guilherme Ottoni[1]
Neil Vachharajani[1] Jonathan Chang[1] David I. August[1]

[1]Department of Computer Science, Princeton University
{bguo, mbridges, ottoni, nvacchar, jcone, august}@princeton.edu
[2]Programming Systems Lab, Intel Corporation
{youfeng.wu, cheng.c.wang}@intel.com

Abstract. Alias analysis, traditionally performed statically, is unsuited for a dynamic binary translator (DBT) due to incomplete control-flow information and the high complexity of an accurate analysis. Whole- program profiling, however, shows that most memory references do not alias. The current technique used in DBTs to disambiguate memory references, instruction inspection, is too simple and can only disambiguate one-third of potential aliases. To achieve effective memory disambiguation while keeping a tight bound on analysis overhead, we propose an efficient heuristic algorithm that strategically selects key memory dependences to disambiguate with runtime checks. These checks have little runtime overhead and, in the common case where aliasing does not occur, enable aggressive optimizations, particularly scheduling. We demonstrate that a small number of checks, inserted with a low-overhead analysis, can approach optimal scheduling, where all false memory dependences are removed. Simulation shows that better scheduling alone improves overall performance by 5%.

1 Introduction

Dynamic Binary Translators (DBTs) are used to provide binary compatibility across platforms. For efficient execution, the translated binary must be re-optimized for the target microarchitecture. This paper focuses on techniques that allow memory disambiguation to be performed in a DBT, enabling advanced optimizations, such as load/store reordering and redundant memory operation elimination, that rely on aliasing information. However, traditional static pointer/alais analysis [1, 2], is expensive both in time and memory, making it unsuitable for DBTs where contention for runtime resources with the program execution itself needs to be kept to a minimum. Additionally, for correctness, the analysis must know all control flows or it becomes overly conservative. Since control flows in DBTs are discovered on the fly as each new branch target is being translated, accurate pointer analysis would require recomputation, taking yet more time that is not available [3–5].

Given the difficulties of performing a full-fledged pointer analysis at runtime, most DBTs, such as Dynamo [3], Transmeta [4], and Daisy [5], do not perform pointer alias analysis except in the form of *instruction inspection*, a simple dependence test

that disambiguates two memory references if they access either different memory regions or their addresses have the same base register and different offsets. While our whole-program profile of the SPECINT2000 benchmarks indicates that 97% of memory reference pairs do not alias, instruction inspection can only disambiguate one-third of them. Without a more sophisticated disambiguation mechanism, the optimizer has to conservatively assume dependences between the other memory references. These false dependences [1] greatly constrain the aggressiveness of various code transformations.

To provide better memory disambiguation for runtime optimizations while keeping a tight control over runtime analysis costs we only attempt to disambiguate specific memory references that actually hide optimization opportunities. This is in contrast to performing pointer analysis on all memory references. In particular, we design a simple heuristic algorithm that precisely selects memory dependences whose removal may result in shortened instruction schedules. It does so without having to recompute the dependence graph and compare the before-and-after schedules. Correctness is guaranteed by inserting runtime checks that dynamically compare the effective addresses of the memory references involved. To maximize the benefit of each runtime check, we perform a light-weight but effective pointer analysis to identify all memory dependences that can be safely removed either directly or indirectly by a single check. For this to work correctly, the runtime check must take into account different offsets of each memory reference, using dynamic address profiles to reduce misspeculation.

We evaluated our technique and experimental results show that only a small number of checks need to be inserted to yield performance gain. Specifically, our technique can remove more than twice as many false memory dependences as does instruction inspection and generate schedules close to the optimal schedules, where all false memory dependences are removed. Finally, this is done with very low analysis overhead.

In summary, the main contributions of this work are:

- An efficient heurisitic algorithm that precisely identifies memory dependences whose removal can benefit scheduling to the greatest extent.
- A light-weight pointer analysis that allows as many dependences as possible to be safely removed by a single runtime check.
- An evaluation of our technique that compares with baseline, instruction inspection and optimial scheduling.

We discuss related work in Section 2. Sections 3 and 4 present the heuristic dependence selection algorithm and the light-weight pointer analysis. Section 5 describes how the test condition for each runtime check is determined. Evaluation methodology and experimental results are discussed in Section 6. Finally, we conclude in Section 7.

2 Related Work

The idea of speculatively disambiguating memory references and relying on runtime tests to guard against misspeculation is not new. The work closest to ours is Nicolau's

[1] In this paper, "false dependence" refers to a dependence that does not occur at runtime, not anti-dependence or output dependence in the traditional data dependence terminology.

run-time disambiguation [6], where the compiler inserts branches that test for aliasing conditions. It relies on trace scheduling to schedule the on-trace path aggressively, assuming that the aliasing conditions are not met, and to insert compensation code in the off-trace path for correctness. This is illustrated by Figure 1. The original code on the left contains a read-after-write (RAW) memory dependence. However, if a check comparing the two addresses is inserted, then in the common case where the addresses are not equal, the load instruction can be moved above the store. Huang et al. [7] describe a similar technique targeting architectures that support conditional execution. Instead of explicit branches, it uses predication to guard the execution of the two paths. Fernandez et al. propose *speculative alias analysis* [8], which is more precise along the hot paths but may not be correct with respect to the whole control flow graph. Any optimization enabled by this analysis requires similar check-and-recovery mechanism.

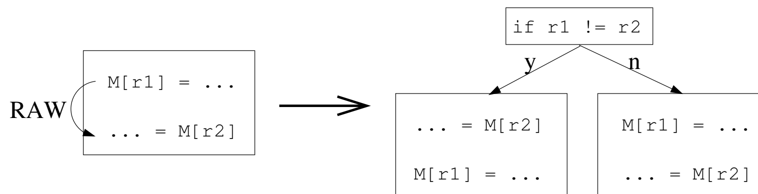

Fig. 1. Example of Runtime Memory Disambiguation

In order to control the code growth resulting from the introduction of extra execution paths, it is important to narrow the set of runtime tests to those that are essential to performance gains. Fernandez et al. [8] does not provide any mechanism to do so. Nicolau [6] skips memory references that can be disambiguated statically with traditional alias analysis and memory references between whom there exist other types of dependences that cannot be removed by runtime disambiguation. Huang et al. [7] uses an iterative heuristic that, after each memory dependence is selected for runtime disambiguation, recomputes the critical path and the estimated execution time before and after removing a dependence for each of the remaining memory dependences. Unlike these two works, both of which are compiler techniques and therefore can afford the cost of a traditional alias analysis or an iterative heuristic, our technique has to meet the much tighter analysis budget in a runtime environment. Not only is our heuristic for selecting critical memory dependence more streamlined and efficient, but also we use light-weight pointer analysis to maximize the coverage of each runtime test.

Data speculation that moves loads above potentially aliasing stores also exists in other DBTs, but relies on special hardware in the target architecture for detecting and recovering from misspeculation. DAISY [5] has a special *load-verify* instruction. Placed at the original position of the speculative load, it reloads the value to compare with the speculatively loaded value and traps to the virtual machine manager if the two values differ. The drawback is that the extra loads executed consume memory bandwidth and energy. Transmeta [4] has a small cache called *alias buffer*, which records the addresses and sizes of speculative loads to compare with later stores for aliases. In this approach, the number of speculative loads is limited by hardware size and false positives may arise as the result of aliasing with unreachable stores. Our approach does not assume

any hardware support and does not suffer from these problems. Neither of these two works performs analysis like ours to select the most beneficial loads for speculation.

3 Critical Memory Dependence Selection

3.1 Preliminary Selection

Before applying the heuristic algorithm to identify memory dependences critical to scheduling, several preliminary steps are taken to prepare a group of candidates.

1. **Trace selection:** We only want to disambiguate memory references in frequently executed code. Hot code identification often comes for free in DBTs as most of them are organized into two phases. The first phase translates blocks of code without optimization and inserts instrumentation to collect execution frequency information. The second phase forms hot regions from frequently executed blocks and applies optimizations to them. Unlike the profiling done in the compilers, which may suffer from the problem of unrepresentative input sets, the profile information collected by DBTs in the first phase is highly relevant to the optimizations done in the second phase. In most DBTs, the hot regions are single entry and multiple exit traces. In our evaluation, the average finishing rate, the probability the trace finishes execution in the last of its constituent blocks, is 88%.
2. **Instruction inspection:** Instruction inspection is performed on each trace to filter out memory reference pairs that definitely do not alias. We then build the dependence graphs of the traces and label each dependence edge with its latency.
3. **Alias profiling:** For those memory references in the traces that cannot be determined to be independent by instruction inspection, instrumentation is inserted to record the effective addresses accessed. The heuristic algorithm will not consider memory reference pairs that actually alias. We find that the aliasing behavior is highly stable throughout the lifetime of a program. That is, a very short inital profiling period yields essentially the same prediction of alias/non-alias as does whole-program profiling. For example, the length of the alias profiling period can be set to end after a trace finishes execution in its last block 50 times. The profiling overhead thus incurred is negligible.
   For the SPECINT2000 benchmarks, true aliases that can be filtered out this way are at most 3%. For other workloads, alias profiling might turn out to be more useful. In addition, the effective addresses collected by alias profiling are also useful later when guiding the determination of appropriate test conditions for the runtime checks. This is discussed in Section 5.

3.2 The Heuristic Algorithm

The goal of the heuristic algorithm is to narrow the number of runtime checks inserted per trace to just 1 or 2 and no more than 3 for the occasional large traces. Our experimental results indicate that this is sufficient to improve scheduling to close-to optimal. Given the small number of memory dependences that are to be removed, the kind of iterative algorithm proposed in [7], which involves recomputation of critical paths and

estimated execution time, is not necessary. After removing only a couple of dependence arcs, we do not expect the memory dependences remaining on the new critial path to be drastically different from what has been there on the original critical path. In addition, we can simply use the latency of each dependence edge to approximate the difference between the execution time of the trace before and after the edge is removed. Based on these reasonings, the basic idea of our heuristic algorithm is to simply pick memory dependences that are responsibile for the largest latency on the original critical paths.

Selecting Critical Base Address Pairs We start by grouping memory instructions according to their *base addresses*. This is done through simple syntactic inspection of the memory operands of each instruction. In the x86 ISA, memory addresses are specified by the expressions $\mathrm{base_reg} + \mathrm{index_reg} * \mathrm{scale} + \mathrm{offset}$, where scale and offset are constants. By base address, we refer to the part of the expression that involves registers, ignoring the constant offset. Memory instructions accessing constant addresses (i.e. no base addresses) are gathered in the same group.

The intuition behind this is the observation that a trace often contains multiple memory references with the same base address but different offsets. If the registers involved in the base address are not redefined in between these references or if it can be proven that the redefinitions always write the same values into those registers, then a single runtime check examining the runtime value of the base address can allow multiple memory dependences to be removed. For example, both the registers EBP (frame pointer) and ESP (stack pointer) are used as the base register and combined with various displacements to access stack locations. In compiler-generated code, stack references with either EBP or ESP as the base register almost never alias. Using a single runtime check that compares the positions pointed to by EBP and ESP and the proper test condition that takes into account all relevant displacements, we can often remove numerous dependences from a trace that cannot otherwise be disambiguated by instruction inspection because the base registers are different. Section 4 describes the analysis needed for proving runtime equality of two occurrences of a base address expression. Section 5.1 discusses how to handles multiple displacements from a base address.

In the next step, the algorithm traverses the dependence graph computed based on instruction inspection to do two things: 1) to identify critical paths, 2) for each pair of base addresses, to sum the latencies of all memory dependence arcs that are false dependences according to the alias profiling and whose source and destination instructions fall into the two groups of base addresses respectively. We use ***Total_Latencies*** to denote this value. Next, the algorithm computes another value similar to ***Total_Latencies***, the only difference being that only memory dependences on the critical paths are considered. We call this value ***Critical_Latencies***. Figure 2 (a) contains a small trace whose dependence graph is shown in Figure 2 (b). The dependence arcs are marked with latencies computed based on the machine model. Arcs with latencies in brackets are memory dependences, the rest are register dependences. Among the four memory instructions, there are three distinct base addresses: EDI, EDX*2, and EBX. In the dependence graph, memory instructions having the same base address are represented with the same symbol. Figure 2 (c) shows the values of ***Total_Latencies*** and ***Critical_Latencies*** for each base address pair.

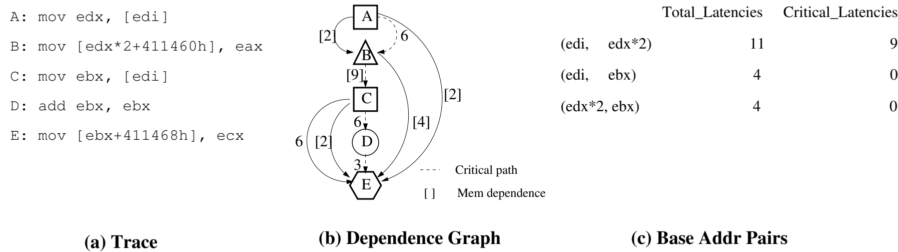

Fig. 2. Example of Target Memory Dependence Selection

The algorithm then selects for runtime disambiguation the pair of base addresses that has the largest non-zero ***Critical_Latencies***. Ties are broken using ***Total_Latencies***. The pair of base addresses with the next largest non-zero ***Critical_Latencies*** is also selected. At this point, for most traces, there are no more base address pairs responsible for latencies on the critical paths. For some very large traces, we allow a third pair to be selected. Beyond three pairs, our experience is that the extra number of runtime checks do not yield substantial performance gains. We refer to the selected base address pairs as ***critical base addresses***. In the example shown in Figure 2, there is only one base address pair (EDI, EDX*2) responsible for latencies on the critical path and therefore it will be selected by the algorithm for runtime disambiguation.

Generating Inputs to Pointer Analysis Because the registers involved in a base address may be redefined within a trace, we need to analyze the trace to determine whether a single runtime check is sufficient to validate assumptions about multiple occurrences of the base address. If not, priority is given to the earliest occurrences in the program order because oftentimes removing one memory dependence gives pointer analysis more accurate information about memory content and thereby helps unravel other memory dependences (details are given in Section 4). The earlier a memory dependence is removed, the more chances there are for it to help eliminate other dependences. For each selected critical base address pair, the heuristic algorithm identifies the earliest pair of memory instructions on the critical path with the corresponding base addresses. These are the inputs to the pointer analysis, which automatically considers the specified memory instruction pairs to be independent.

As an optimization, if there is an even earlier pair, though not on the critical path, it may be returned instead by the algorithm, but only if the registers involved in the address expressions are not redefined between this pair and the earliest pair on the critical path. This way we guarantee that the latter is always disambiguated, hence the critical path shortened. In Figure 2, the earliest memory dependence corresponding to the selected base address pair (EDI, EDX*2) is the edge $A \to B$, which is not on the critical path.

4 Light-Weight Pointer Analysis

Given the critical base address pairs returned by the heuristic algorithm, the goal of the pointer analysis is to identify two kinds of memory dependences:

Directly covered dependences Memory references whose base addresses (which syntactically may or may not look the same as the critical base addresses) are guaranteed to evaluate to the same runtime value as the critical base addresses.

Indirectly covered dependences Memory references that, though not accessing memory locations pointed to by the critical base addresses, may still be disambiguated as a result of more accurate pointer information when some false memory dependences have been removed.

The analysis achieves this by computing symbolically the set of possible values for each register and each memory location touched within a trace. Amme et al. design a intra-procedural data dependence analysis for assembly code by symbolic evaluation [9]. However, their algorithm does not keep track of memory contents and therefore loses crucial information and accuracy.

The key to the success of our analysis is not just to prove non-aliases, but to infer must-aliases such that information about the content of a memory location can be propagated from one memory reference to another. The fact that the traces are single-entry greatly increases the rate at which must-aliases can be proven since every use has exactly one reaching definition for both registers and memory locations. In addition, this control-flow property also keeps the analysis extremely light-weight because the size of any symbolic value set is always exactly 1 due to the absence of merge points.

Notice that this analysis only needs to be performed on traces for which the heuristic algorithm returns at least some memory dependences to recommend for runtime disambiguation. Also notice that we could have used this more sophisticated analysis in place of instruction inspection in the preliminary dependence selection phase to filter out more non-aliases. However, without some dependences assumed to be removed by runtime checks, the analysis is too constrained by inaccurate information about memory contents to offer significant benefit over instruction inspection. On top of that, the analysis will have to be run on all traces. Therefore we use instruction inspection instead, which is simpler and cheaper.

The remainder of this section first walks through a small example to show how the analysis works and then gives the formal definition of the analysis.

4.1 Walking through an Example

In the trace shown in Figure 2, there are five pairs of ambiguous memory references: $A \rightarrow B$, $A \rightarrow E$, $B \rightarrow C$, $B \rightarrow E$, and $C \rightarrow E$. We show that all of them can be removed by inserting one runtime check.

The pointer analysis receives from the memory dependence selection algorithm the input instruction pair (A, B), which can be assumed to be independent since a runtime check will be inserted to compare the values of EDI and EDX*2. The same check also directly removes the dependence between B and C because EDI remains unchanged between A and C. Although syntactically the address referenced by instruction E has nothing to do with EDI or EDX, interestingly the remaining three dependences involving it can still be eliminated in the presence of the check. Since A and B do not alias, we know that the content of the location pointed to by EDI is not overwritten by B, therefore the values loaded by A and C must be the same, that is, EBX $==$ EDX right after

```
//r1     = base1 + index1 * scale1 + offset1
//r2     = base2 + index2 * scale2 + offset2
//result = base  + index  * scale + offset

base   := r1
index  := r2
scale  := a
offset := b

if (base2 == NULL && index2 == NULL)
    index  := NULL
    offset += offset2 * a
else if (index2 == NULL)
    index  := base2
    offset += offset2 * a
else if (base2 == NULL)
    index  := index2
    scale  *= scale2
    offset += offset2 * a

if (index1 == index || index1 == NULL  || index == NULL)
    base   := base1
    scale  += scale1
    offset += offset1

if (base == index)
    base := NULL
    scale++
```

Fig. 3. Pseudo-code for Computing $r1 + r2 * a + b$

C. After symbolically executing instruction D, EBX $==$ EDX*2, hence the symbolic address referenced by E is EDX*2 + 411468h with EDX*2 as base. This means that B and E definitely do not alias since they access the same base address with different offsets. The dependence between $A \rightarrow E$ and $C \rightarrow E$ are removed because their base addresses can be compared by the runtime check.

4.2 Symbolic Pointer Values

We use the same expression, $\text{base_reg} + \text{index_reg} * \text{scale} + \text{offset}$, to represent all symbolic values, pointers or non-pointers. Each register value in the expression is a pair $(\text{reg_name}, \text{def_site})$, where def_site is the id of the instruction that writes the value into the register. Either base_reg or index_reg can be omitted (we say that their value is NULL). The rule for arithmetics on the symbolic values is described by the pseudo-code in Figure 3, which computes $r1 + r2 * a + b$ where $r1$ and $r2$ are themselves symbolic expressions. It merges the two symbolic expressions if it can, otherwise it gives up and returns $r1 + r2 * a + b$.

4.3 Analysis Algorithm

The algorithm finishes in one pass over the instructions in the trace starting from the entry. At each memory instruction, it compares the symbolic address with those of all previous memory instructions to see whether aliases exist. Therefore the worst case complexity of the algorithm is quadratic in the number of instructions.

```
//Instruction t is a memory reference
addr_t := Inst_Addrs(t)

for each memory instruction s before t
    addr_s := Inst_Adds(s)
    if (may_alias(addr_s, addr_t))
        record dependence s->t
        if (is_store(t))
            Mem_Values(addr_s) := NULL

if (is_load(t))
    r := dest_reg(t)
    content_t := Mem_Values(addr_t)
    if (content_t != NULL)
        Reg_Values(r) := content_t
    else
        Reg_Values(r) := (r, t)
        Mem_Values(addr_t) := Reg_Values(r)
```

Fig. 4. Pseudo-code for Analyzing Memory Instructions

The algorithm maintains the following data structures.

1. ***Reg_Values*** – Maps each register to a symbolic expression describing the current value held in that register.
2. ***Mem_Values*** – Maps each symbolic memory address to a symbolic expression describing the current value stored at that address. If the analysis cannot infer any information about the content of that memory location, it will map the symbolic address to the value NULL.
3. ***Inst_Addrs*** – Maps each memory instruction to the symbolic address it references.

Figure 4 contains the pseudo-code for actions taken at memory instructions. At other instructions, the analysis simply does symbolic evaluation based on the semantics of the instruction.

The symbolic address of each memory instruction is compared with those of all memory instructions that come before it in the trace. Non-alias is determined if the base address parts of the two symbolic addresses are the same and the offsets are different. Must-alias is determined if both the base addresses and the offsets are the same. If the memory instruction is a store, the analysis removes the contents of all aliasing symbolic addresses recorded in the table ***Mem_Values*** and changes them to NULL. This is because the store might write to any of these aliasing locations, destroying the values held in there. If the memory instruction is a load, the analysis looks up the symbolic address up in ***Mem_Values***. If it is mapped to a non-NULL value, this means that there is a previous must-alias instruction for which the analysis has recorded what value is in the memory location right after the instruction and this information has not been destroyed by any subsequent aliasing store. In this case, the destination register of the load can assume the value recorded in ***Mem_Values***. Otherwise, the analysis can say nothing about the value loaded into the destination register, either because the value has been destroyed by aliasing stores or because the value is a live-in through memory. In this case, the analysis simply records in ***Mem_Values*** that the content of the memory location is whatever value that is currently in the destination register.

5 Inserting Runtime Checks

Runtime checks are inserted in the trace and scheduled together with other instructions. Instructions that do not depend on the checks can be scheduled past them, no compensation code is needed on the off-trace path. Instructions dependent on the checks but with no side-effects, i.e. loads and their uses, can also be scheduled past the checks. We rely on the trace scheduling algorithm [10] to insert proper compensation code at the split points.

5.1 Determining Test Conditions

Each memory reference is characterized by an address and a reference size, which together specify a range of memory addresses $[\mathrm{address}, \mathrm{address}+\mathrm{size})$. A runtime check needs to test for the disjoint-ness of two memory ranges. Let $range1 = [a, b)$ and $range2 = [c, d)$, either $range1$ is below $range2$, which is captured by the condition $b <= c$, or $range1$ is above $range2$, which is captured by the condition $d <= a$. For each pair of critical base addresses $(\mathrm{base_a}, \mathrm{base_b})$, we separate the memory dependences directly covered by it into two groups to differentiate between these two situations. It turns out that just like the aliasing behavior, the relative positions of any two memory references are also highly stable throughout the lifetime of the program. This is not surprising as the two memory references may access different data structures whose locations in memory are fixed. As such, profiling can provide good guidance on deciding which group a memory dependence should go to. Indeed, we use the actual addresses recorded in the initial alias profiling phase for this purpose. Let $a \rightarrow b$ be a dependence directly covered by $(\mathrm{base_a}, \mathrm{base_b})$ and suppose that instruction a's base address is $\mathrm{base_a}$ and instruction b's base address is $\mathrm{base_b}$. If the actual address range of a is below the actual address range of b according to the profile, then we put $a \rightarrow b$ in group I, otherwise we put it in group II.

Two checks are then generated, one for each group. Within each group, the check needs to be able to accommodate all the different offsets from a single base address. To do this, we introduce the concept of an ***extended range***, which contains all ranges relevant to a base address within either group. Suppose that there are n dependences in group I (or group II) and that the n instructions with $base_a$ as base address are described by the (symbolic_address, size) pairs $(\mathrm{base_a}+\mathrm{offset_1}, \mathrm{size_1})$, ..., and $(\mathrm{base_a}+\mathrm{offset_n}, \mathrm{size_n})$. Then the extended range corresponding to $\mathrm{base_a}$ in this group is
$[\mathrm{base_a} + min(\mathrm{offset_1}, ..., \mathrm{offset_n}), \mathrm{base_a} + max(\mathrm{offset_1} + \mathrm{size_1}, ..., \mathrm{offset_n} + \mathrm{size_n}))$.
The extended range for $base_b$ can be computed in the same fashion.

For example, in Figure 5, there are three memory dependences covered by same critical base address pair: no base address (instruction A, B, and D) and `edx` (instruction C). All memory references have a size of 4 bytes. The dependences $A \rightarrow C$ and $B \rightarrow C$ belong to group I as the effective address range of A and B are below that of C, and the dependence $C \rightarrow D$ belongs to group II. The extended range covering both A and B is $[301280h, 411464h + 4)$ and the extended range covering C is $[\texttt{edx} + 8, \texttt{edx} + 8 + 4)$. Therefore the test condition we generate for group I is $411464h + 4 <= \texttt{edx} + 8$. Similarly the test condition for group II is $\texttt{edx} + 8 + 4 <= 62013Ch$.

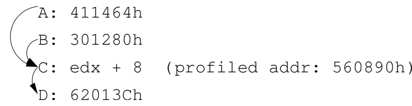

Fig. 5. Determining Test Conditions

6 Evaluation

6.1 Experimental Framework

We implemented selective runtime disambiguation algorithm in the Star Dynamic Binary Translator (StarDBT), a DBT framework currently being developed inside Intel for 32-bit x86. We evaluated our technique on an in-order VLIW simulator with stall-on-load semantics. The simulator models a 6-issue processor with 2 memory read ports, 2 memory write ports, 4 integer units, 1 floating point unit, and 1 branch unit. The configuration of the memory hierarchy is given in Table 1. The memory behavior of StarDBT itself is not simulated since the execution time spent in StarDBT code is a small fraction of the execution time of the entire program. Execution time is computed by summing of the static cycle counts generated from the instruction schedules and the miss penalties reported by a cache simulator.

Level	Write Policy	Allocation Policy	Floating Point Bypass	Associativity	Size	Latency
L1	Write through	Read-only alloc	Yes	4-way	16KB	1 cycle hit
L2	Write back	Write alloc	No	8-way	128KB	3 cycle hit
L3	Write back	Write alloc	No	12-way	3MB	10 cycle hit / 100 cycles miss

Table 1. Memory Hierarchy

The simulation is done online while StarDBT is translating and executing the program. StarDBT inserts instrumentation before each memory instruction to record the actual addresses in a data structure. Whenever the data structure is filled up, StarDBT jumps out of the execution of the program and transfers control to the cache simulator, which then simulates the memory accesses in the order specified by the instruction schedules. Figure 6 contains the flow chart of the simulation process within StarDBT. After a trace is formed, the first schedule is computed based on dependence information generated from instruction inspection. Next, alias profiling is conducted, during which period the memory accesses of the trace are simulated based on the first schedule. At the end of alias profiling, our heuristic algorithm and light-weight pointer analysis are performed and a second schedule is generated if any runtime check is to be inserted. From that point on till the end of the program execution, memory accesses are simulated using the second schedule. The benchmarks we used is the SPECINT2000 benchmarks with ref input.

6.2 Precision Evaluation

Precision in this context is how finely our technique controls where to apply selective disambiguation and where to spend analysis effort. We also look at the misspeculation

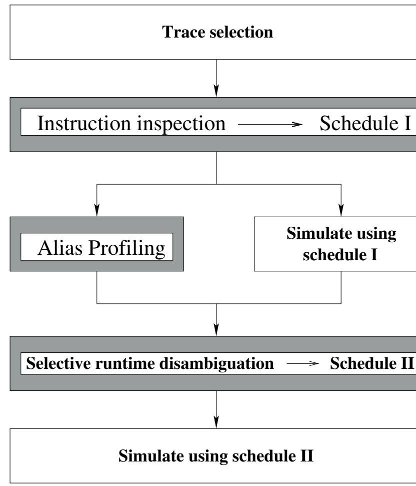

Fig. 6. Flow Chart of Simulation in StarDBT

Benchmark	#Traces	#Selected Traces	#Optimized Traces	#Checks	Misspeculation
164.gzip	1558	364	362	636	- %
175.vpr	1220	366	349	552	0.01%
176.gcc	14924	2251	2164	3124	- %
181.mcf	174	50	50	76	0.01%
186.crafty	1431	175	173	263	1.51%
197.parser	2566	376	347	645	1.55%
252.eon	739	215	211	390	5.75%
253.perlbmk	9299	3093	2913	5217	0.36%
254.gap	1691	287	281	444	0.38%
255.vortex	3956	1734	1713	3990	- %
256.bzip2	963	247	241	359	0.85%
300.twolf	931	311	302	548	0.48%

Table 2. Precision Evaluation

rate of the runtime checks to see whether the test conditions accurately capture the frequent cases.

In Table 2, the column "#Selected Traces" refers to traces for which the heuristic algorithm reports beneficial critical dependences, the column "#Optimized Traces" refers to traces that, with runtime checks inserted, indeed have schedules shorter than their original schedules. There are two interesting points. First, on average the selected traces are only 26% of the total traces. The rest simply do not have memory dependences on the critical path. This could happen if the memory dependence is not the only type of dependence between two instructions and there exists a chain of other dependences whose total latency is larger than the latency of the memory dependence. From this point, the heuristic algorithm is precise in that no check is ever inserted where it cannot possibly improve scheduling. Second, out of the selected traces, almost all (97%) have improved schedules with runtime checks inserted. The reason why some traces may fail to have shorter schedules is that not enough dependences are removed by the runtime checks. For example, there may be two critical paths of equal lengths in the original dependence graph and the checks can only remove dependences on one path, or too few dependences are removed to make up for the overhead of the runtime

checks themselves. From this point, the heuristic algorithm is also precise because the critical base addresses it selects are such that the runtime checks for these addresses almost always cover enough dependences to yield actual performance gains, hence no work subsequently done in the pointer analysis and rescheduling is wasted.

For the traces that do have improved schedules, we go ahead and insert the runtime checks and recovery code. On average, only about 1.7 checks are inserted per optimized trace. If averaged over all traces that make up the program, about 0.4 checks are inserted per trace. Assuming pessimistically that with each check inserted the entire trace has to be duplicated, this translates to a rough estimation of a 40% code growth over the original translated binary. In reality, however, the code growth should be much smaller because checks are often inserted not at the entry of the trace but in the middle, therefore only the tail of the trace needs to be duplicated.

Misspeculation happens when a runtime check fails either because the initial alias profiling did not accurately predict the aliasing behavior of the whole program or because the test conditions fail to characterize all cases of disjoint-ness in their attempt to cover multiple displacements from base addresses. The misspeculation rate is measured in StarDBT by prolonging the initial alias profiling phase to span the entire execution of the program and evaluating the test conditions on the effective addresses collected by profiling. The last column in Table 2 gives the misspeculation rate as the percentage of the number of misspeculations over the total number of runtime checks performed dynamically.

6.3 Impact on Scheduling

To evaluate the impact of removed memory dependences on the quality of the instruction schedules, we compare the schedules generated with no disambiguation at all, with instruction inspection, and with our technique respectively to the optimal schedules, where all false memory dependences are removed. The quality of the schedules is measured as slowdown from the optimal schedules in static cycle counts. As we can see from Table 7, without any memory disambiguation, the schedules generated assuming all memory references alias are really bad, with a geometric mean slowdown of 24.5%. With instruction inspection, the slowdown is reduced to 7.7%. And finally, our technique can almost close the gap and match the optimal scheduling (1.3% slowdown) just by inserting a small number of runtime checks.

6.4 Speedup from Improved Scheduling

We compare the actual performance gains, obtained respectively from instruction inspection only and from our technique, over baseline where no memory disambiguation is performed at all. The same online cache simulation mechanism is used to compute the execution time of baseline and instruction inspection. As shown in Table 3, our technique can disambiguate more than twice as many dependences as does instruction inspection, which amounts to 73% of all false memory dependences. This translates to a 5% increase in performance gains over baseline as the result of improved instruction scheduling alone. If combined with other optimizations such as redundant memory operation elimination and register promotion, we expect even bigger improvement.

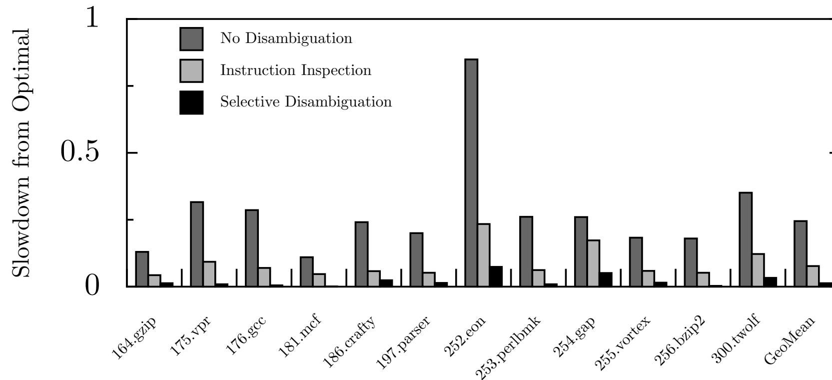

Fig. 7. Comparing with Optimal Scheduling

Benchmark	Instruction Inspection		Selection Disambiguation	
	Deps Removed	Speedup	Deps Removed	Speedup
164.gzip	42%	7.7%	78%	10.3%
175.vpr	30%	16.9%	80%	23.3%
176.gcc	35%	16.8%	69%	21.9%
181.mcf	34%	5.6%	79%	9.9%
186.crafty	31%	14.2%	47%	18.9%
197.parser	40%	12.4%	80%	15.5%
252.eon	34%	33.3%	69%	42.0%
253.perlbmk	32%	15.8%	76%	20.0%
254.gap	17%	6.9%	57%	16.6%
255.vortex	34%	10.4%	87%	14.2%
256.bzip2	47%	13.2%	75%	17.2%
300.twolf	40%	16.9%	84%	23.5%
Average	35%	14.2%	73%	19.3%

Table 3. Performance Gains over Baseline

6.5 Analysis Overhead

We measured the time spent in our analysis, which corresponds to the shaded boxes in Figure 6. It is a function of the number of traces and the size of the traces on which the light-weight pointer analysis and rescheduling are performed. Table 4 shows the analysis time together with the execution time of the benchmarks in StarDBT without our technique on a 3.2GHz Xeon with 2.5MB of cache and 2G of memory. [1] The analysis overhead is extremely low compared to the program execution time, so the performance gained through our technique will not be offset by the overhead of the technique itself.

7 Conclusion and Future Work

In this paper, we present a technique designed to provide sophisticated memory disambiguation in a dynamic binary translator at low cost. It precisely selects memory dependences whose removal by runtime disambiguation can result in shortened schedules. Simple analysis is applied to allow as many dependences to be removed by one runtime

[1] StarDBT does not yet run on in-order machines that we want to evaluate our technique on, hence the speedup through our technique could not be measured in the same way.

Benchmark	Execution Time (sec)	Analysis Time (sec)
164.gzip	125	0
175.vpr	129	1
176.gcc	122	3
181.mcf	104	0
186.crafty	112	0
197.parser	144	1
252.eon	92	5
253.perlbmk	166	5
254.gap	70	1
255.vortex	105	9
256.bzip2	129	0
300.twolf	191	1

Table 4. Analysis Time

check as possible. We also use profile guidance to trim the actual test conditions of the runtime checks. Our experiments demonstrate that selective runtime memory disambiguation almost doubles the number of memory dependences removed by instruction inspection and improves the overall performance by 5% just from scheduling. In the future, we will investigate other optimizations such as redundant memory operation elimination and register promotion, which can make use of the disambiguation offered by our technique for further improvements.

References

1. B.-C. Cheng and W. W. Hwu, “Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 57–69, 2000.
2. W. Landi and B. G. Ryder, “A safe approximate algorithm for interprocedural pointer aliasing,” in *Proceedings of the ACM SIGPLAN ’92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.
3. V. Bala, E. Deusterwald, and S. Banerjia, “Transparent dynamic optimization,” Tech. Rep. HPL-1999-77, Hewlett Packard Labs, June 1999.
4. J. C. Dehnert, B. K. Grant, and J. P. Banning, “The transmeta code morphing software: using speculation, recovery and adaptive retranslation to address real-life challenges,” in *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pp. 15–24, March 2003.
5. K. Ebcioglu and E. R. Altman, “DAISY: Dynamic compilation for 100% architectural compatibility,” in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.
6. A. Nicolau, “Run-time disambiguation: Coping with statically unpredictable dependences,” *IEEE Transactions on Computers*, vol. 38, pp. 663–678, May 1989.
7. A. S. Huang, G. Slavengurg, and J. P. Shen, “Speculative disambiguation: A compilation technique for dynamic memory disambiguation,” *ACM SIGARCH Computer Architecture News Archive*, vol. 22, pp. 200–210, April 1994.
8. M. Fernandez and R. Espasa, “Speculative alias analysis for executable code,” in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pp. 222–231, September 2002.
9. W. Amme, P. Braun, and E. Zehendner, “Data dependence analysis of assembly code,” Tech. Rep. 3764, INRIA, Rocquencourt, France, September 1999.
10. J. A. Fisher, “Trace scheduling: A technique for global microcode compaction,” *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.