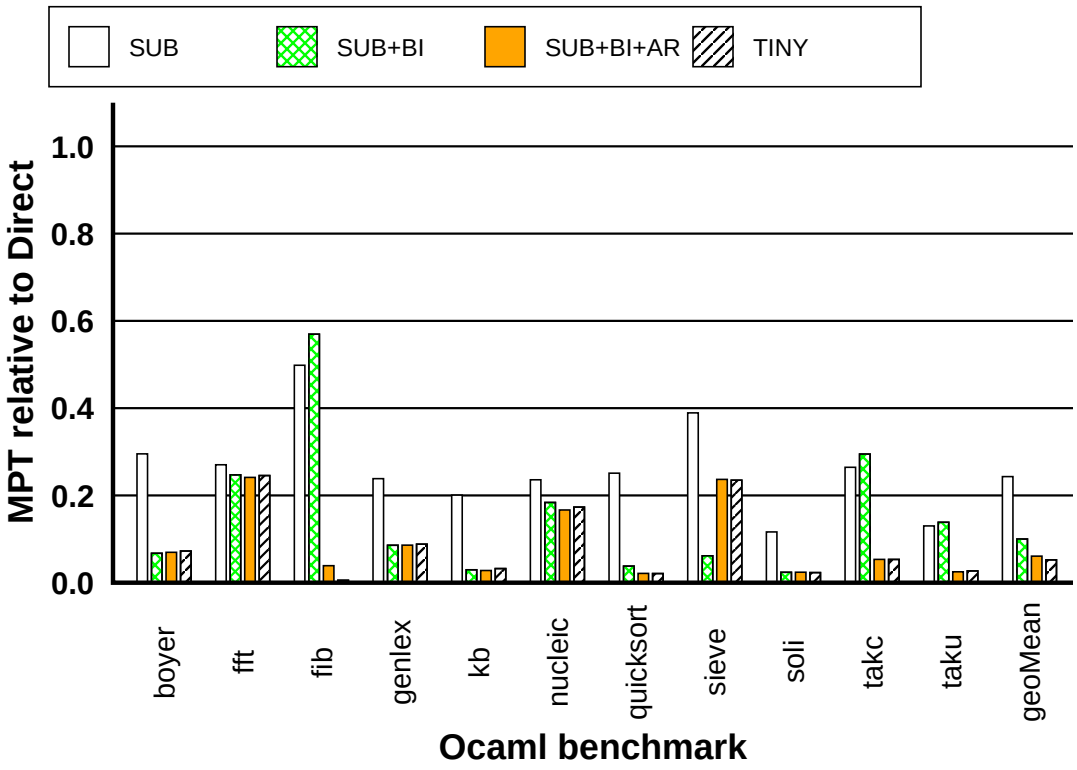
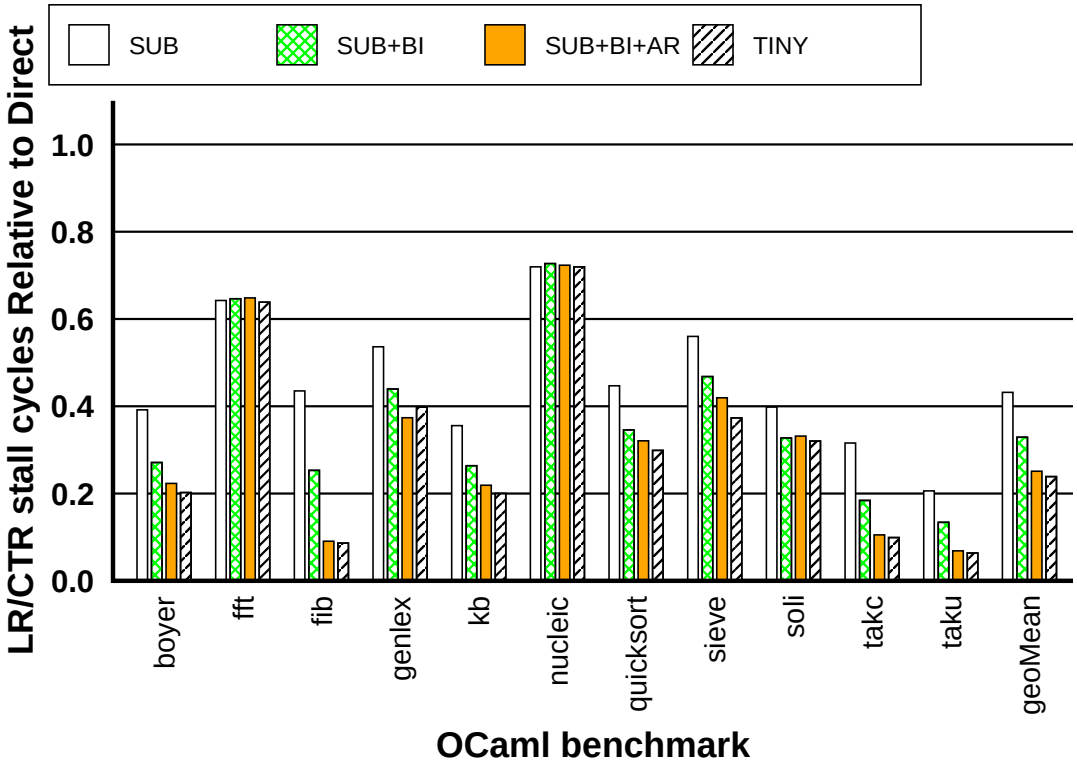
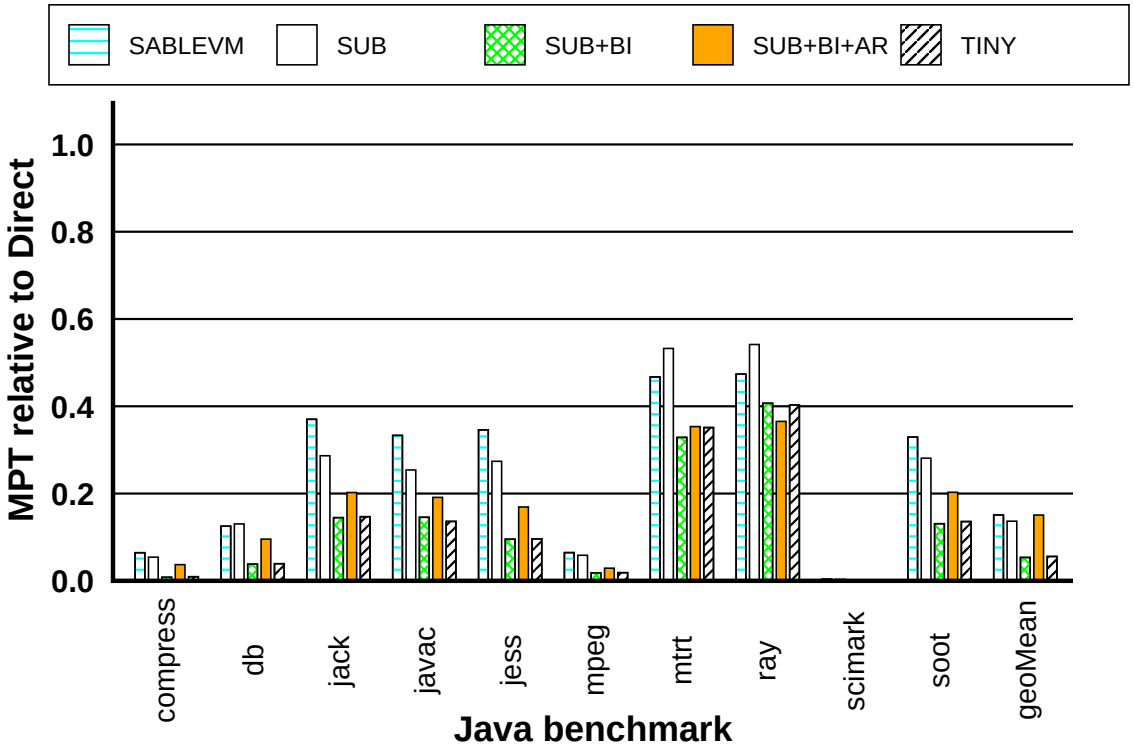



(a) Pentium 4 Mispredicted Taken Branches

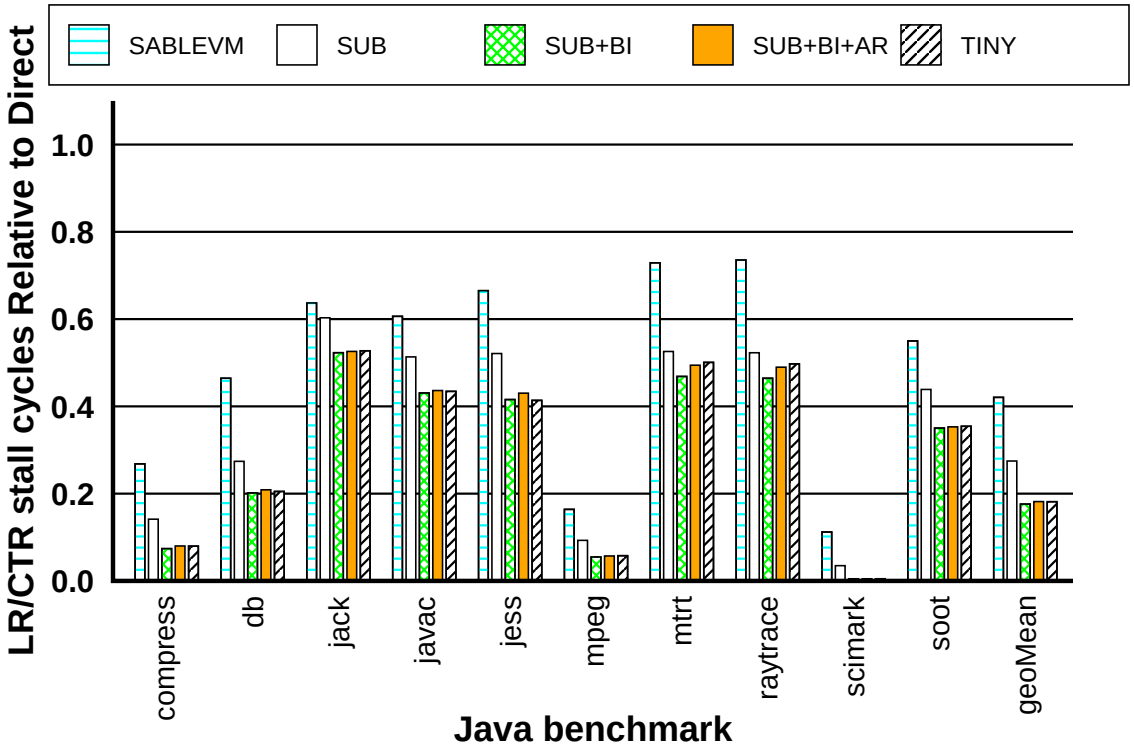



(b) PPC 7410 LR/CTR stall cycles

Figure 5.1: OCaml Pipeline Hazards Relative to Direct Threading

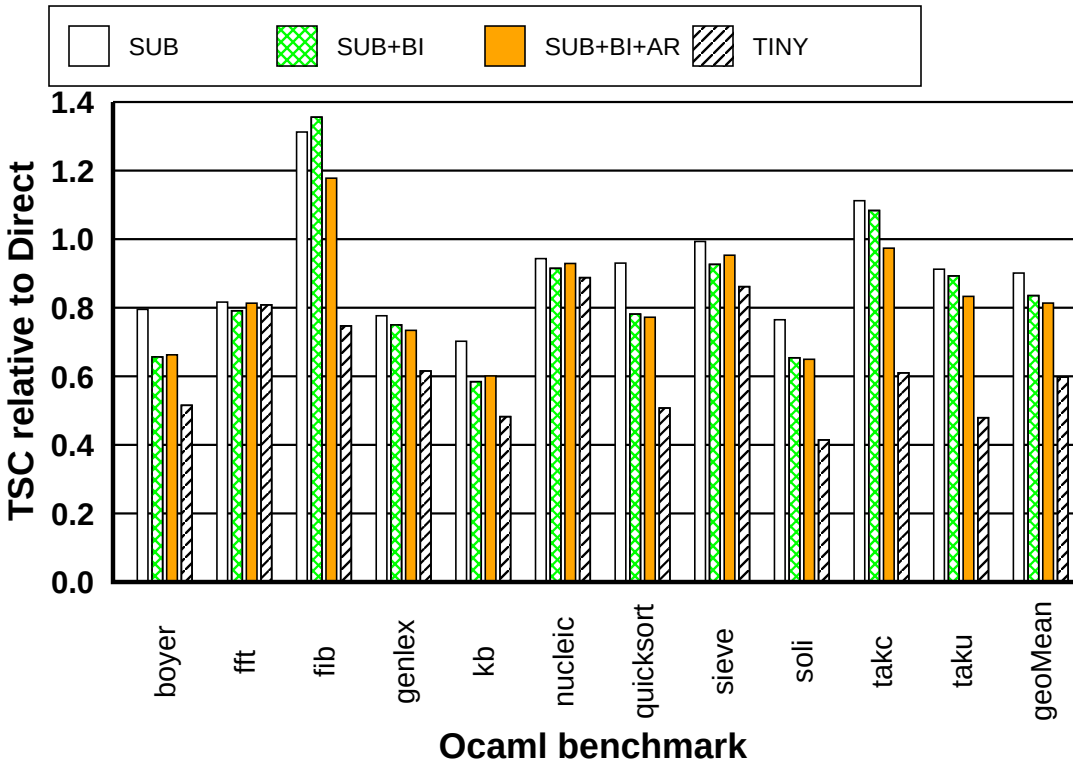



(a) Pentium 4 Mispredicted Taken Branches

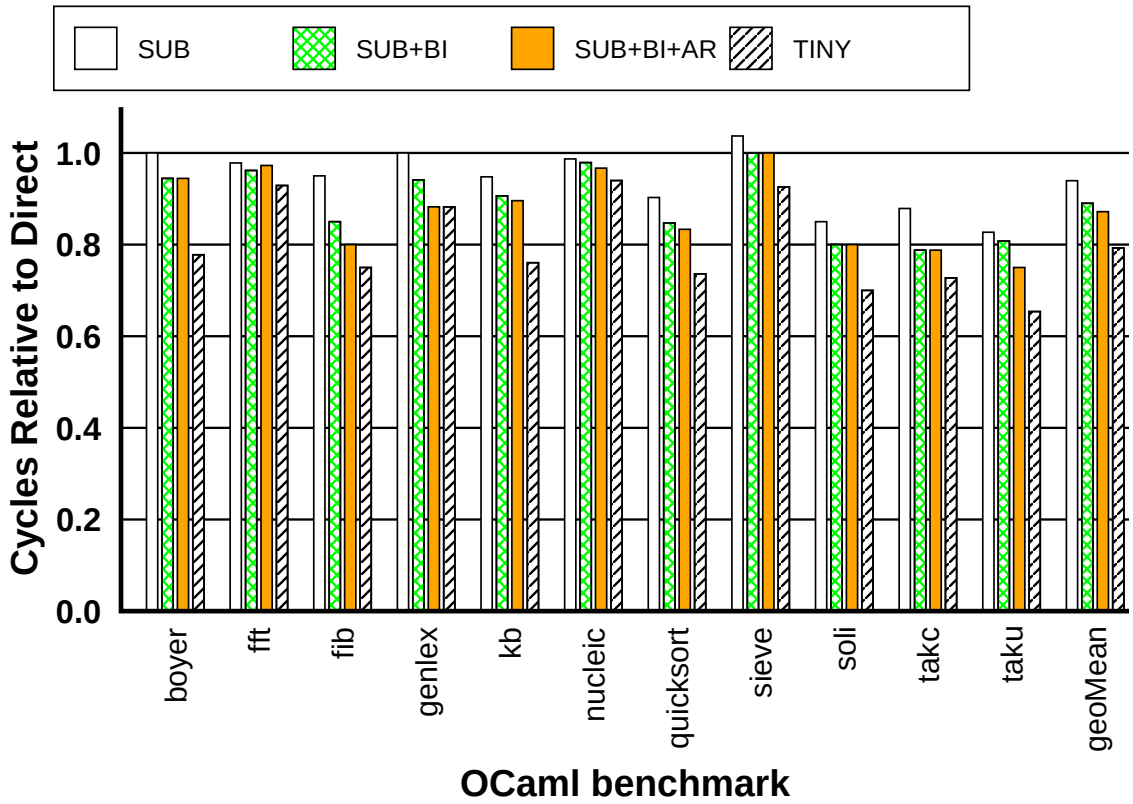



(b) PPC7410 - LR/CTR stall cycles

Figure 5.2: Java Pipeline Hazards Relative to Direct Threading

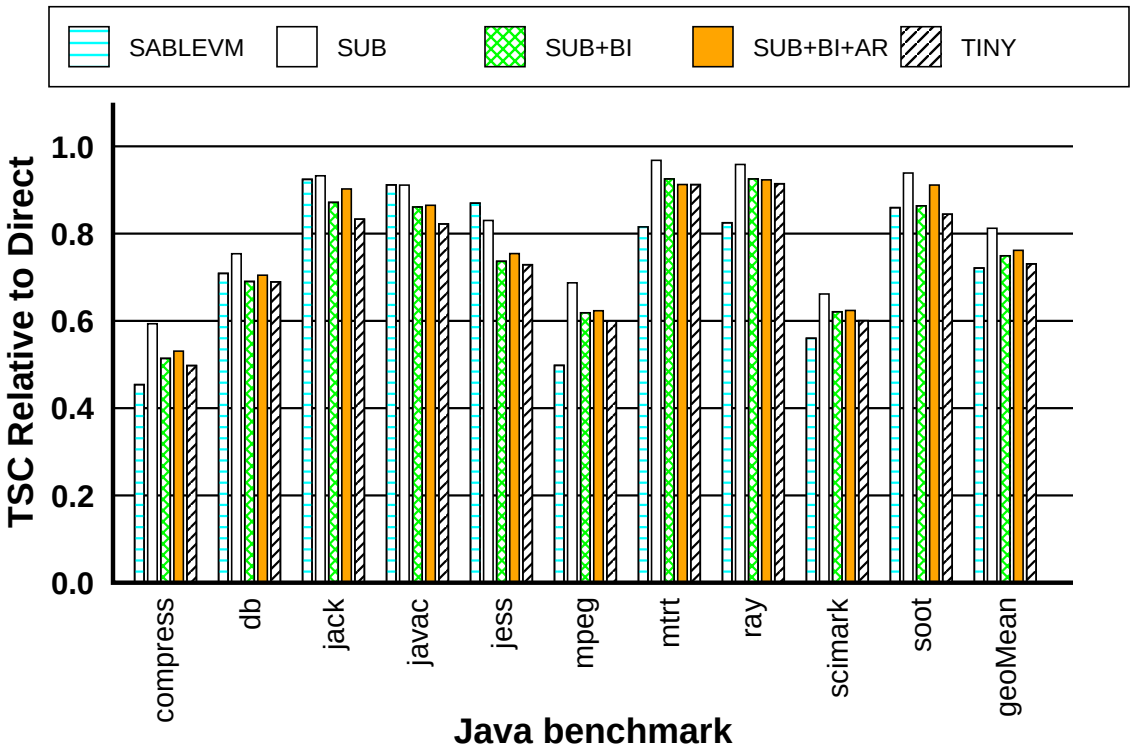



(a) Pentium 4

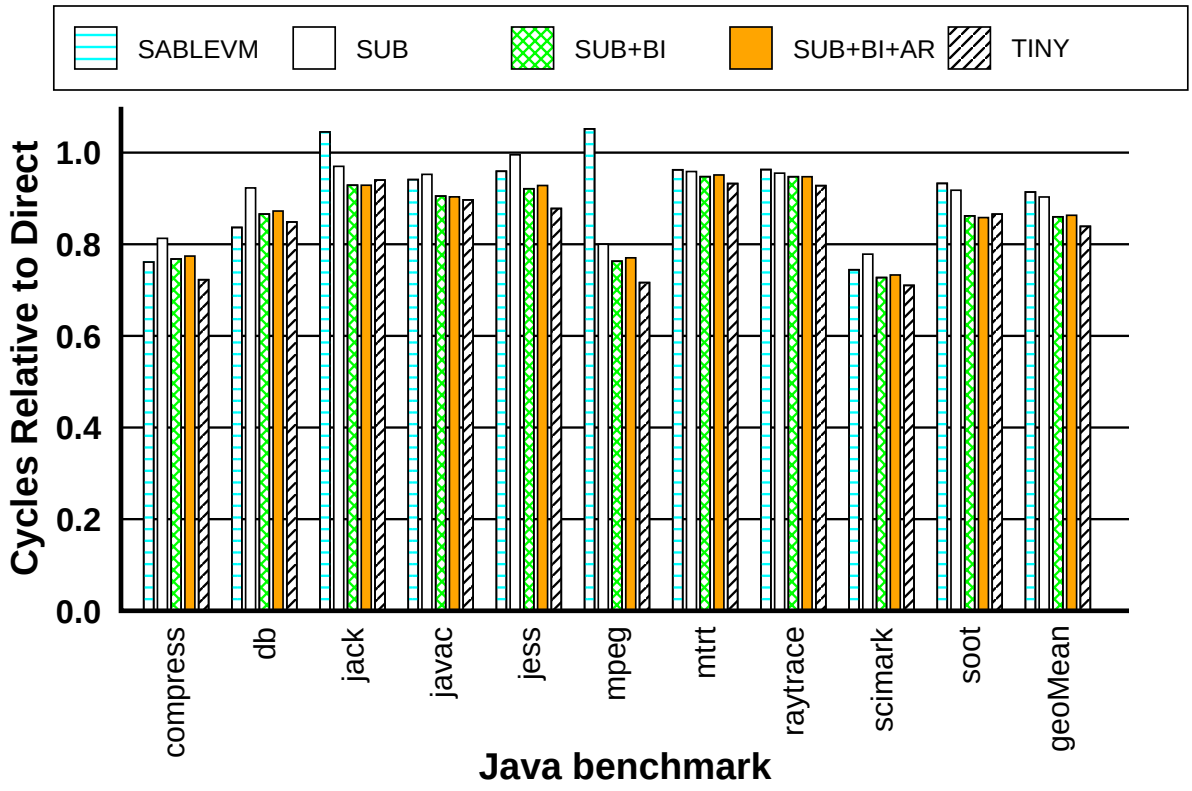



(b) PPC7410

Figure 5.3: OCaml Elapsed Time Relative to Direct Threading

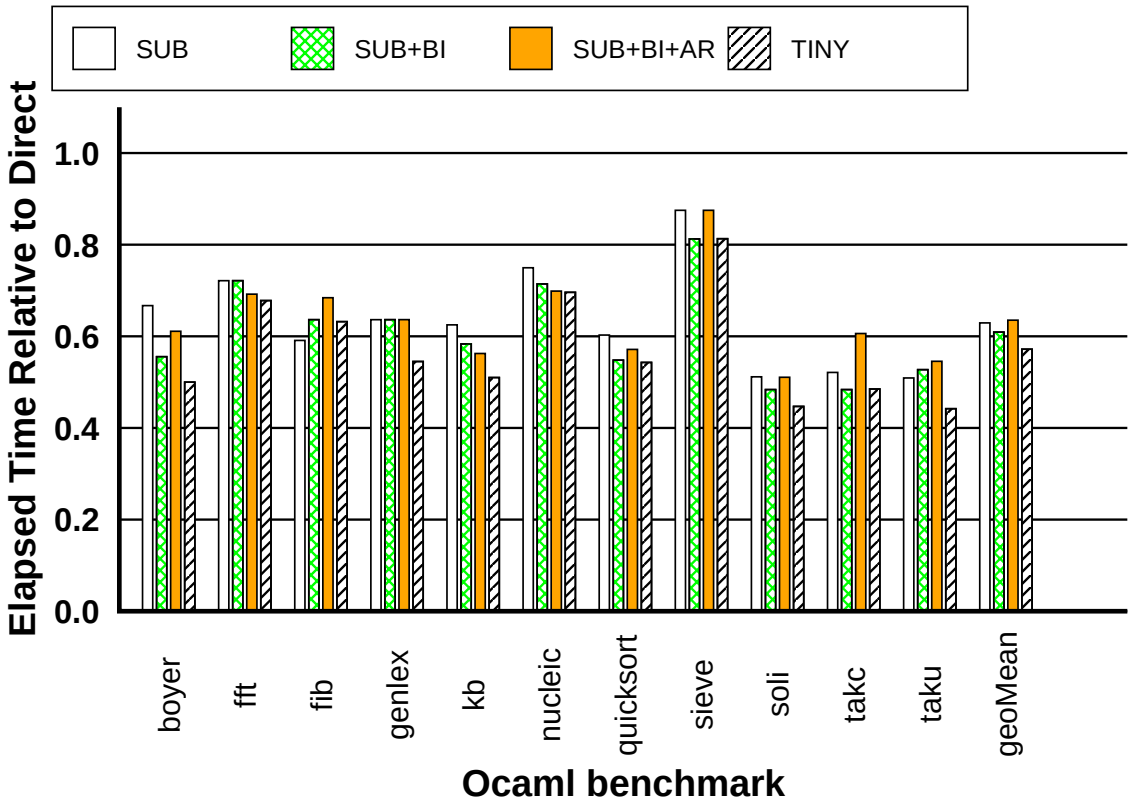



(a) Pentium 4



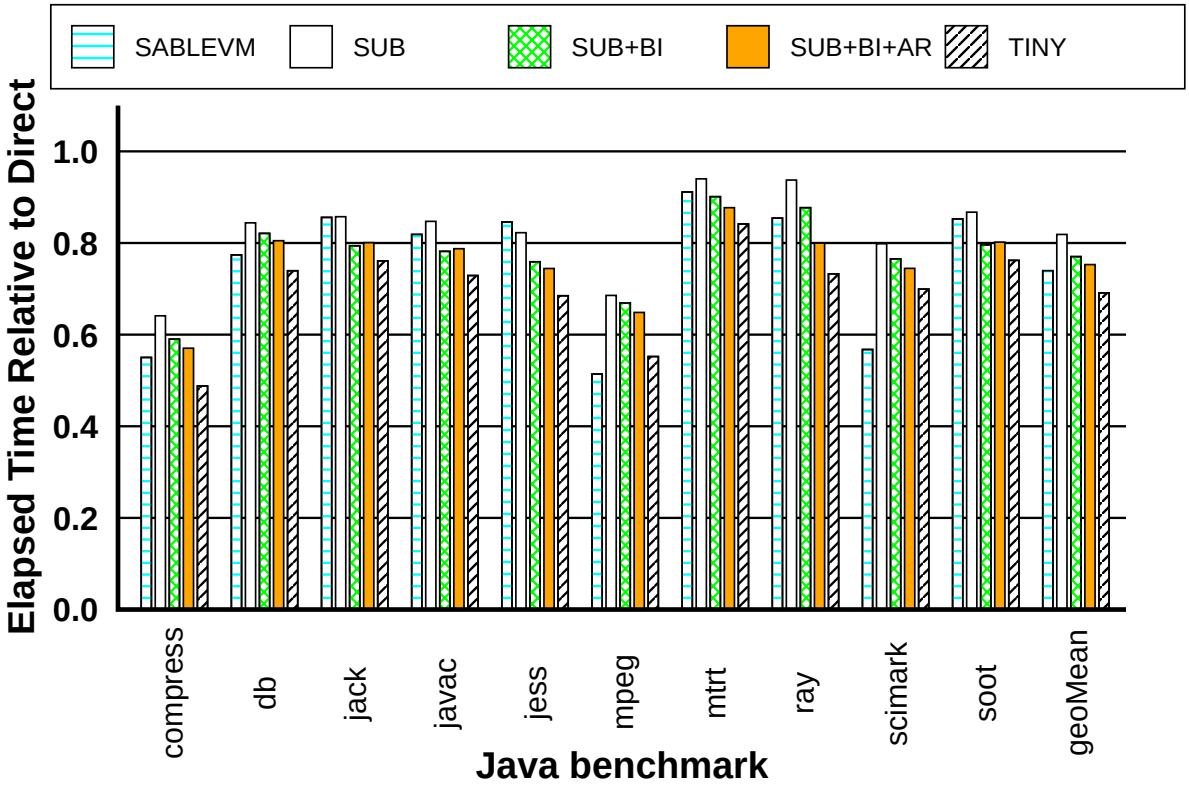

(b) PPC 7410

Figure 5.4: SableVM Elapsed Time Relative to Direct Threading




s

(a) OCaml PPC970 elapsed (real) seconds




(b) SableVM PPC970 elapsed (real) seconds

Figure 5.5: PPC970 Elapsed Time Relative to Direct Threading

OCaml VM is about 19% lower for context threading than direct threading on P4, 9% lower on PPC7410, and 39% lower on the PPC970. For SableVM, SUB+BI+AR, compared with direct threading, runs about 17% faster on the PPC7410 and 26% faster on both the P4 and PPC970. Although we cannot measure the cost of LR/CTR stalls on the PPC970, the greater reductions in execution time are consistent with its more deeply-pipelined design (23 stages vs. 7 for the PPC7410).

Across interpreters and architectures, the effect of our techniques is clear. Subroutine threading has the single largest impact on elapsed time. Branch inlining has the next largest impact eliminating an additional 3–7% of the elapsed time. In general, the reductions in execution time track the reductions in branch hazards seen in Figures 5.1 and 5.2. The longer path length of our dispatch technique are most evident in the OCaml benchmarks `fib` and `takc` on the P4 where the improvements in branch prediction (relative to direct threading) are minor. These tiny benchmarks compile into unique instances of a few virtual instructions. This means that there is little or no sharing of BTB slots between instances and hence fewer mispredictions.

The effect of apply/return inlining on execution time is minimal overall, changing the geometric mean by only $\pm 1\%$ with no discernible pattern. Given the limited performance benefit and added complexity, a general deployment of apply/return inlining does not seem worthwhile. Ideally, one would like to detect heavy recursion automatically, and only perform apply/return inlining when needed. We conclude that, for general usage, subroutine threading plus branch inlining provides the best trade-off.

We now demonstrate that context-threaded dispatch is complementary to inlining techniques.

## 5.3 Inlining

Inlining techniques address the context problem by replicating bytecode bodies and removing dispatch code. This reduces both instructions executed and pipeline hazards. In this section we

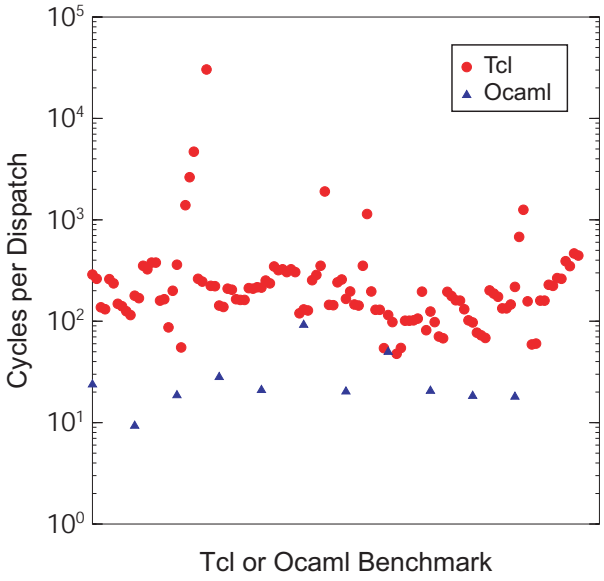



Figure 5.6: Reproduction of [77, Figure 1] showing cycles run per virtual instructions dispatched for various Tcl and OCaml benchmarks .

performance of subroutine threaded OCaml on an UltraSPARC III[4]. As shown in the figure, subroutine threading speeds up OCaml on the UltraSPARC by about 13%. In contrast, the geometric mean of 500 Tcl benchmarks speeds up only by only 5.4% [77].

Another issue raised by the Tcl implementation was that about 12% of the 500 program benchmark suite slowed down. Very few of these dispatched more than 10,000 virtual instructions. Most were tiny programs that executed as little as a few dozen dispatches. This suggests that for programs that execute only a small number of virtual instructions, the load time overhead of generating code in the CTT may be too high.

### 5.4.2 Context Threading and Profiling

Our original scheme for extending our context threaded interpreter with a JIT was to detect hot paths of the virtual program by generating calls to profiling instrumentation amongst the dispatch code in the CTT. We persevered for some time with this approach, and successfully

[4]We leveraged Vitale's Tcl infrastructure, which only runs on Sparc, to implement subroutine threading. Thus, to compare to Tcl we ported our subroutine threaded OCaml to Sparc also.

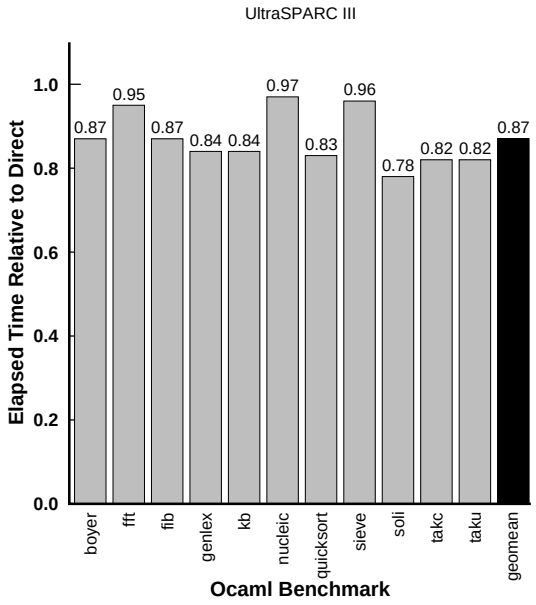



Figure 5.7: Elapsed time of subroutine threading relative to direct threading for OCaml on UltraSPARC III.

implemented a system that identified traces [80]. The resulting implementation, though efficient, was fragile and required the generation of more machine specific code for profiling than we considered desirable. In the next chapter we describe a much more convenient approach based on dispatch loops.

### 5.4.3 Development using SableVM

SableVM is a very well engineered interpreter. For instance, SableVM's infrastructure for identifying un-relocatable virtual instruction bodies made implementing our TINY inlining experiment simple. However, its heavy use of `m4` and `cpp` macros, used to implement multiple dispatch mechanisms and achieve a high degree of portability, makes debugging awkward. In addition, our efforts to add profiling instrumentation to context threading made many changes that we subsequently realized were ill-advised. Hence, we decided to start from clean sources. For the next stage of our experiment, our trace-based JIT, we decided to abandon SableVM in

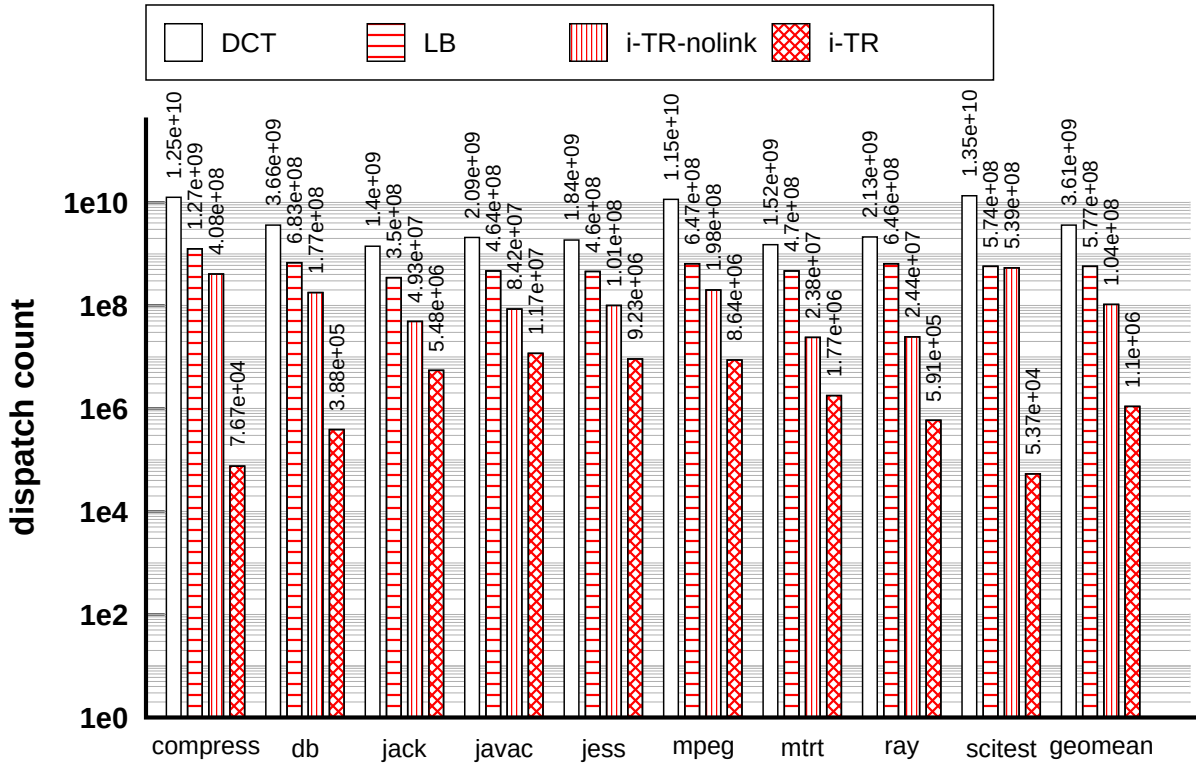



Figure 7.1: Number of dispatches executed vs region shape. The y-axis has a logarithmic scale. Numbers above bars, in scientific notation, give the number of regions dispatched. The X axis lists the SPECjvm98 benchmarks in alphabetical order.

## 7.2 Effect of region shape on dispatch

In this section we report data obtained by modifying Yeti's instrumentation to keep track of how many virtual instructions are executed from each region body and how often region bodies are dispatched. These data will help us understand to what extent execution remains in the code cache for differently shaped regions of the program.

For a JIT to be effective, execution must spend most of its time in compiled code. We can easily count how many virtual instructions are executed from interpreted traces and so we can calculate what proportion of all virtual instructions executed come from traces. For `jack`, traces account for 99.3% of virtual instructions executed. For all the remaining benchmarks, traces account for 99.9% or more.

A remaining concern is how often execution enters and leaves the code cache. In our

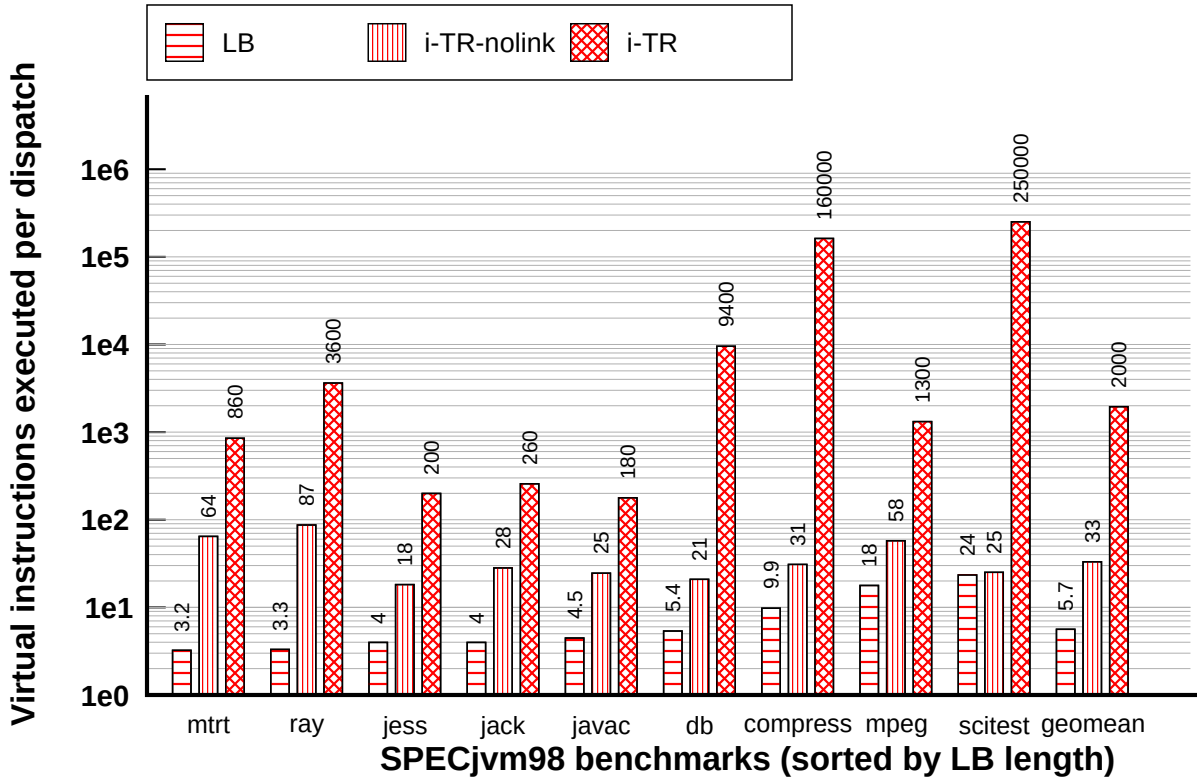



Figure 7.2: Number of virtual instructions executed per dispatch for each region shape. The y-axis has a logarithmic scale. Numbers above bars are the number of virtual instructions executed per dispatch (rounded to two significant figures). SPECjvm98 benchmarks appear along X axis sorted by the average number of instructions executed by a LB.

system, execution enters the code cache whenever a region body is called from a dispatch loop. It is an easy matter to instrument the dispatch loops to count how many iterations occur, and hence how many dispatches are made. These numbers are reported by Figure 7.1. The figure shows how direct call threading (DCT) compares to linear blocks (LB), interpreted traces with no linking (i-TR-nolink) and linked interpreted traces (i-TR). Note that the y-axis has a logarithmic scale.

DCT dispatches each virtual instruction body individually, so the DCT bars on Figure 7.1 report how many virtual instructions were executed by each benchmark. For each benchmark, the ratio of DCT to LB shows the dynamic average linear block length (e.g., for `compress` the average linear block executed $1.25 \times 10^{10}/1.27 \times 10^{9} = 9.9$ virtual instructions). In general, the height of each bar on Figure 7.1 divided by the height of the DCT bar gives the

average number of virtual instructions executed per dispatch of that region shape. Figure 7.2 also presents the same data in terms of virtual instructions executed per dispatch, but sorts the benchmarks along the x axis by the average LB length. Hence, for compress, the LB bar shows 9.9 virtual instructions executed on the average.

Scientific benchmarks appear on the right of Figure 7.2 because they tend to have longer linear blocks. For instance, the average block in `scitest` has about 24 virtual instructions whereas `javac`, `jess` and `jack` average about 4 instructions. Comparing the geometric mean across benchmarks, we see that LB reduces the number of dispatches relative to DCT by a factor of 6.3. On long basic block benchmarks, we expect that the performance of LB will approach that of direct threading for two reasons. First, fewer trips around the dispatch loop are required. Second, we showed in Chapter 5 that subroutine threading is better than direct threading for linear regions of code.

Traces do predict paths taken through the program. The rightmost cluster on Figure 7.2 show that, even without trace linking (i-TR-nolink), the average trace executes about 5.7 times more virtual instructions per dispatch than a LB. The improvement can be dramatic. For instance `javac` executes, on average, about 22 virtual instructions per trace dispatch. This is much longer than its dynamic average linear block length of 4 virtual instructions. This means that for `javac`, on the average, the fourth or fifth trace exit is taken. Or, putting it another way, for `javac` a trace typically correctly predicts the destination of 5 or 6 virtual branches.

This behavior confirms the assumptions behind our approach to handling virtual branch instructions in general and the design of interpreted trace exits in particular. We expect that most of the trace exits, four fifths in the case of `javac`, will not exit. Hence, we generate code for interpreted trace exits that should be easily predicted by the processor's branch history predictors. In the next section we will show that this improves performance, and in Section 7.5 we show that it also reduces branch mispredictions.

Adding trace linking completes the interpreted trace (i-TR) technique. Trace linking makes the greatest single contribution, reducing the number of times execution leaves the trace cache

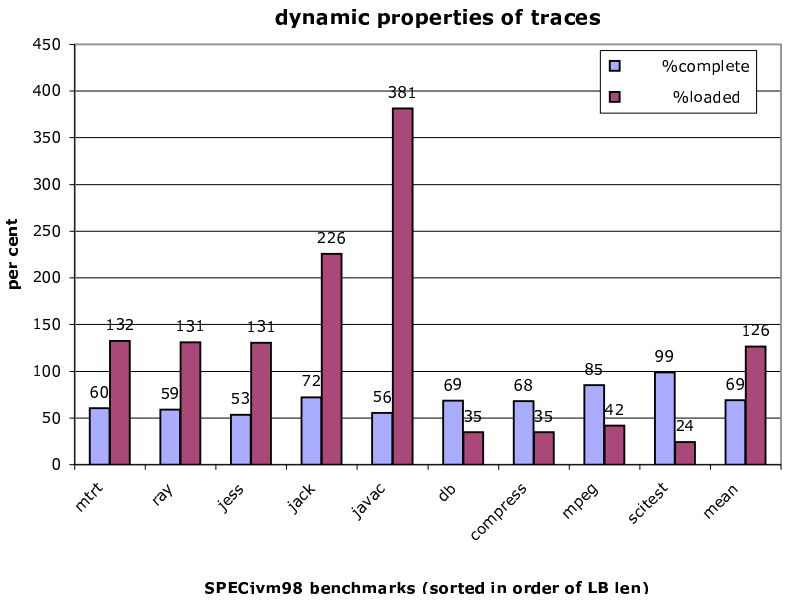



Figure 7.3: Percentage trace completion rate as a proportion of the virtual instructions in a trace and code cache size for as a percentage of the virtual instructions in all loaded methods. For the SPECjvm98 benchmarks and scitest.

by between one and 3.7 *orders of magnitude*. Trace linking has so much impact because it links traces together around loops. A detailed discussion of how inner loops depend on trace linking appears in Section 6.4.3.

Although this data shows that execution is overwhelmingly from the trace cache, it gives no indication of how effectively code cache memory is being used by the traces. A thorough treatment of this, like the one done by Bruening and Duesterwald [11], is beyond the scope of this thesis. Nevertheless, we can relate a few anecdotes based on data that our profiling system already collects.

Figure 7.3 describes two aspects of traces. First, in the figure, the %complete bars report the extent to which traces typically complete, measured as a percentage of the virtual instructions in a trace. For instance, for `raytrace`, the average trace exit occurs after executing 59% of the virtual instructions in the trace. Second, the %loaded bars report the size of the traces in the code cache as a percentage of the virtual instructions in all the loaded methods. For raytrace we see that the traces contain, in total, 131% of the code in the underlying loaded methods.

We observe that for an entire run of the `scitest` benchmark, all generated traces contain only 24% of the virtual instructions contained in all loaded methods. This is a good result for traces, suggesting that a trace-based JIT needs to compile fewer virtual instructions than a method-based JIT. Also, we see that for `scitest`, the average trace executes almost to completion, exiting after executing 99% of the virtual instructions in the trace. This is what one would expect for a program that is dominated by inner loops with no conditional branches – the typical trace will execute until the reverse branch at its end.

On the other hand, for `javac` we find the reverse, namely that the traces bloat the code cache – almost four *times* as many virtual instructions appear in traces than are contained in the loaded methods. In Section 7.5 we shall discuss the impact of this on the instruction cache. Nevertheless, traces in `javac` are completing only modestly less than the other benchmarks. This suggests that `javac` has many more hot paths than the other benchmarks. What we are not in a position to measure at this point is the temporal distribution of the execution of the hot paths.

## 7.3 Effect of region shape on performance

In this section we report the elapsed time required to execute each benchmark. One of our main goals is to create an architecture for a high level machine that can be gradually extended from a simple interpreter to a high performance JIT augmented system. Here, we evaluate the performance of various stages of Yeti's enhancement from a direct call-threaded interpreter to a trace based mixed-mode system.

Figure 7.4 shows how performance varies as differently shaped regions of the virtual program are executed. The figure shows elapsed time relative to the unmodified JamVM distribution, which uses direct-threaded dispatch. The raw performance of unmodified JamVM and TR-JIT is given in Table 7.1. The first four bars in each cluster represent the same stage of Yeti's enhancement as those in Figure 7.1. The fifth bar, TR-JIT, gives the performance of Yeti

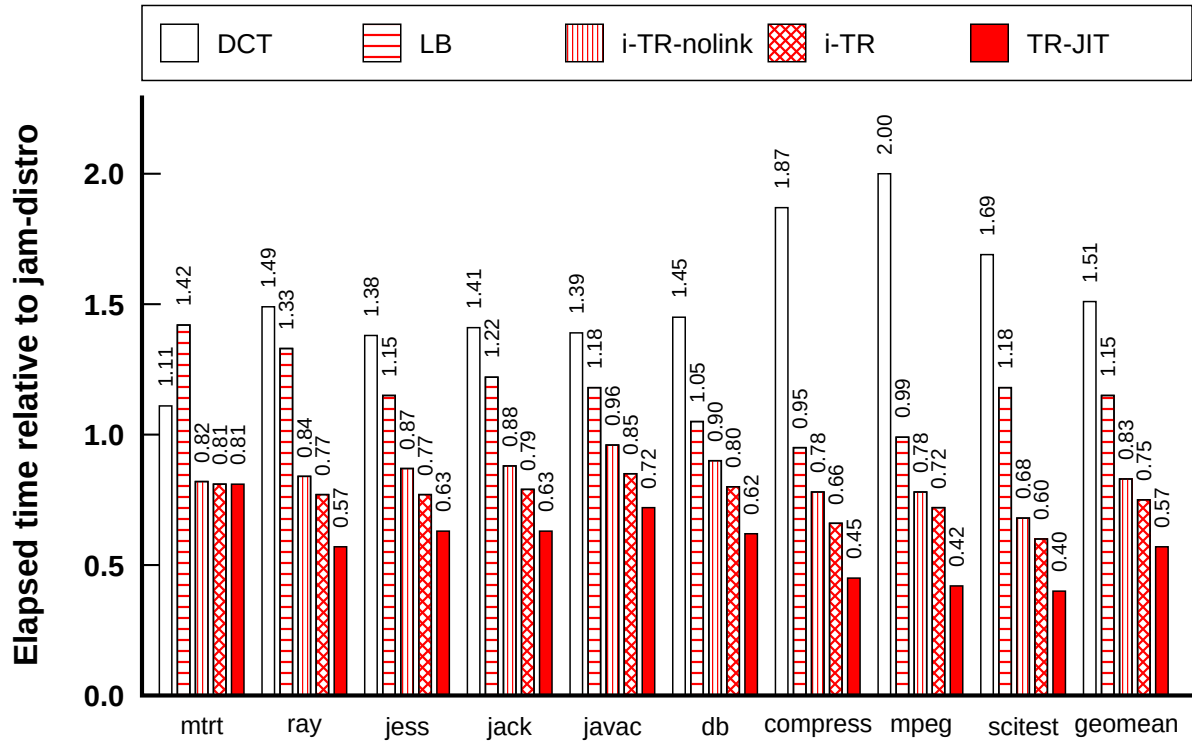



Figure 7.4: Performance of each stage of Yeti enhancement from DCT interpreter to trace-based JIT relative to unmodified JamVM-1.3.3 (direct-threaded) running the SPECjvm98 benchmarks (sorted by LB length).

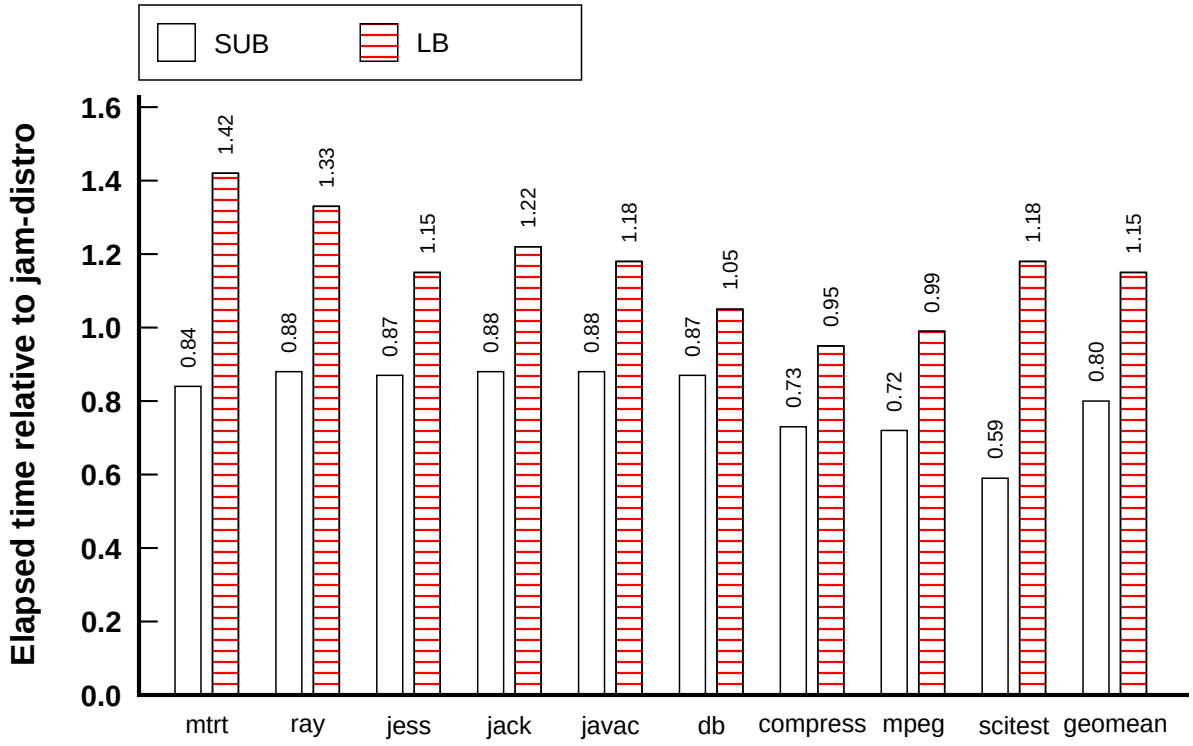



Figure 7.5: Performance of Linear Blocks (LB) compared to subroutine-threaded JamVM-1.3.3 (SUB) relative to unmodified JamVM-1.3.3 (direct-threaded) for the SPECjvm98 benchmarks.

with our JIT enabled.

**Direct Call Threading** Our simplest technique, direct call threading (DCT) is slower than JamVM, as distributed, by about 50%.

Although this seems serious, we note that many production interpreters are not direct threaded but rather use the slower and simpler switch threading technique. When JamVM is configured to run switch threading we we find that its performance is within 1% of DCT. This suggests that the performance of DCT is well within the useful range.

**Linear Blocks** As can be seen on Figure 7.4, Linear blocks (LB) run roughly 30% faster than DCT, matching the performance of direct threading for benchmarks with long basic blocks like `compress` and `mpeg`. On the average, LB runs only 15% more slowly than direct threading.

The region bodies identified at run time by LB are very similar to the code generated by

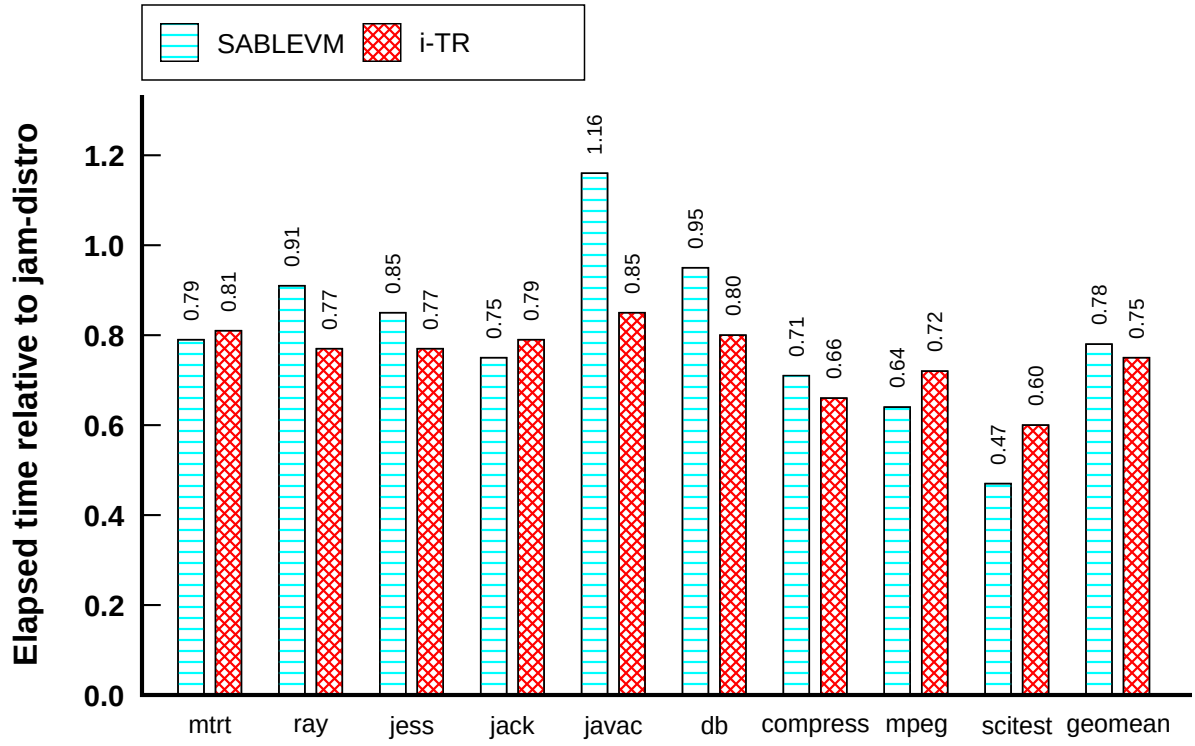



Figure 7.6: Performance of JamVM interpreted traces (i-TR) and selective inlined SableVM 1.1.8 relative to unmodified JamVM-1.3.3 (direct-threaded) for the SPECjvm98 benchmarks.

subroutine threading (SUB) at load time so one might expect the performance of the two techniques to be the same. However, as shown by Figure 7.5 LB is, on the average, about 43% slower.

This is because virtual branches are much more expensive for LB. In SUB, the virtual branch body is called from the CTT[1], then, instead of returning, it executes an indirect branch directly to the destination CTT slot. In contrast, in LB a virtual branch instruction sets the vPC and returns to the dispatch loop to call the destination region body. In addition, each iteration of the dispatch loop must loop up the destination body in the dispatcher structure (through an extra level of indirection compared to SUB).

**Interpreted Traces** Just as LB reduces dispatch and performs better than DCT, so link-disabled interpreted traces (i-TR-nolink) further reduce dispatch and run 38% faster than LB.

[1]See Section 3.6

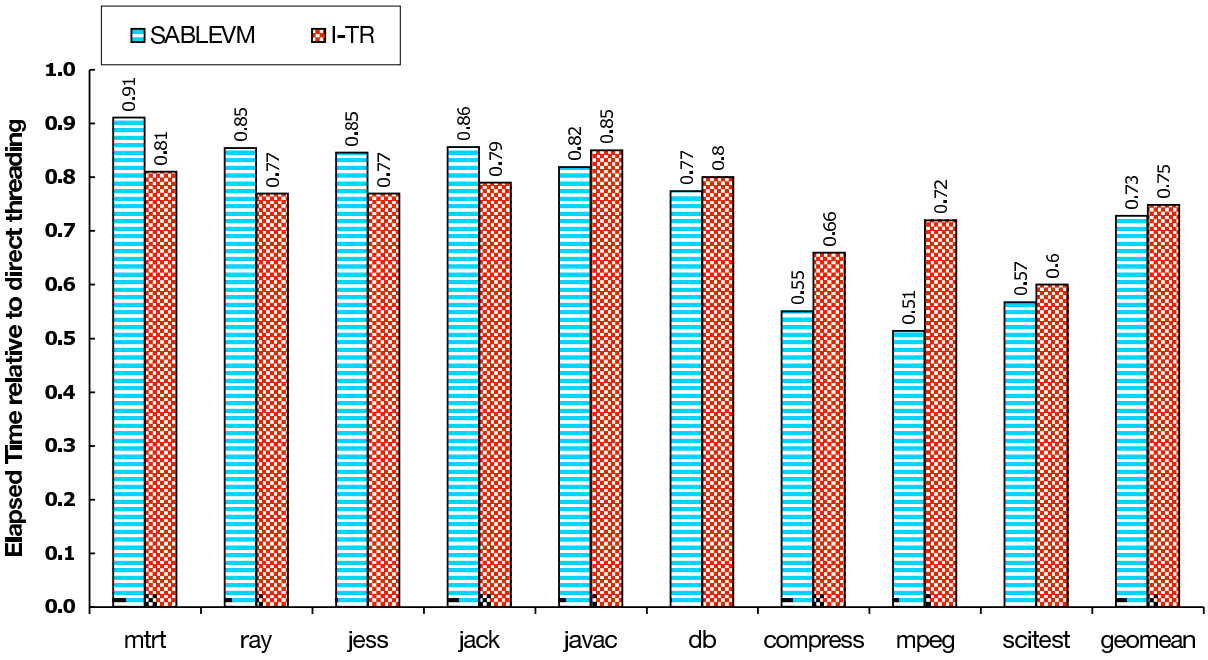



Figure 7.7: Performance of JamVM interpreted traces (i-TR) relative to unmodified JamVM-1.3.3 (direct-threaded) and selective inlined SableVM 1.1.8 relative to direct threaded SableVM version 1.1.8 for the SPECjvm98 benchmarks.

Interpreted traces implement virtual branch instructions better than LB or SUB. As described in Section 6.4.1, i-TR generates a trace exit for each virtual branch. The trace exit is implemented as a direct conditional branch that is not taken when execution stays on trace. As we have seen in the previous section, execution typically remains on trace for several trace exits. Thus, on the average, i-TR replaces costly indirect calls (from the dispatch loop) with relatively cheap not-taken direct conditional branches. Furthermore, the conditional branches are fully exposed to the branch history prediction facilities of the processor.

Trace linking, though it eliminates many more dispatches, achieves only a modest further speed up because the specialized dispatch loop for traces is much less costly than the generic dispatch loop that runs LB.

We compare the performance of selective inlining, as implemented by SableVM, and interpreted traces in two different ways. First, in Figure 7.6, we compare the performance of both techniques relative to the same baseline, in this case JamVM with direct threading. Second, in Figure 7.7, we show the speedup of each VM relative to its own implementation of

direct threading, that is, we show the speedup of i-TR relative to JamVM direct threading and selective inlining relative to SableVM direct threading.

Overall, Figure 7.6 shows that i-TR and SableVM perform almost the same with i-TR about 3% faster than selective inlining. SableVM wins on programs with long basic blocks, like `mpeg` and `scitest`, because selective inlining eliminates dispatch from long sequences of simple virtual instructions. However, i-TR wins on shorter block programs like `javac` and `jess` by improving branch prediction. Nevertheless, Figure 7.7 shows that selective inlining results in a 2% larger speedup over direct threading for SableVM than i-TR. Both techniques result in very similar overall effects even though i-TR is focused on improving virtual branch performance and selective inlining on eliminating dispatch within basic blocks.

Subroutine threading again emerges as a very effective interpretation technique, especially given its simplicity. SUB runs only 6% more slowly than i-TR and SableVM.

The fact that i-TR runs exactly the same runtime profiling instrumentation as TR-JIT makes it qualitatively a very different system than SUB or SableVM. SUB and SableVM are both tuned interpreters that generate a small amount of code at load time to optimize dispatch. Neither includes any profiling infrastructure. In contrast to this, i-TR runs all the infrastructure needed to identify hot traces at run time. As we shall see in Section 7.5, the improved virtual branch performance of interpreted traces has made it possible to build a profiling system that runs faster than most interpreters.

**JIT Compiled traces** The rightmost bar in each cluster of Figure 7.4 shows the performance of our best-performing version of Yeti (TR-JIT). Comparing geometric means, we see that TR-JIT is roughly 24% faster than interpreted traces. Despite supporting only 50 integer and object virtual instructions, our trace JIT improves the performance of integer programs such as `compress` significantly. With our most ambitious optimization, of virtual method invocation, TR-JIT improved the performance of `raytrace` by about 35% over i-TR. `Raytrace` is written in an object-oriented style with many small methods invoked to access object fields.

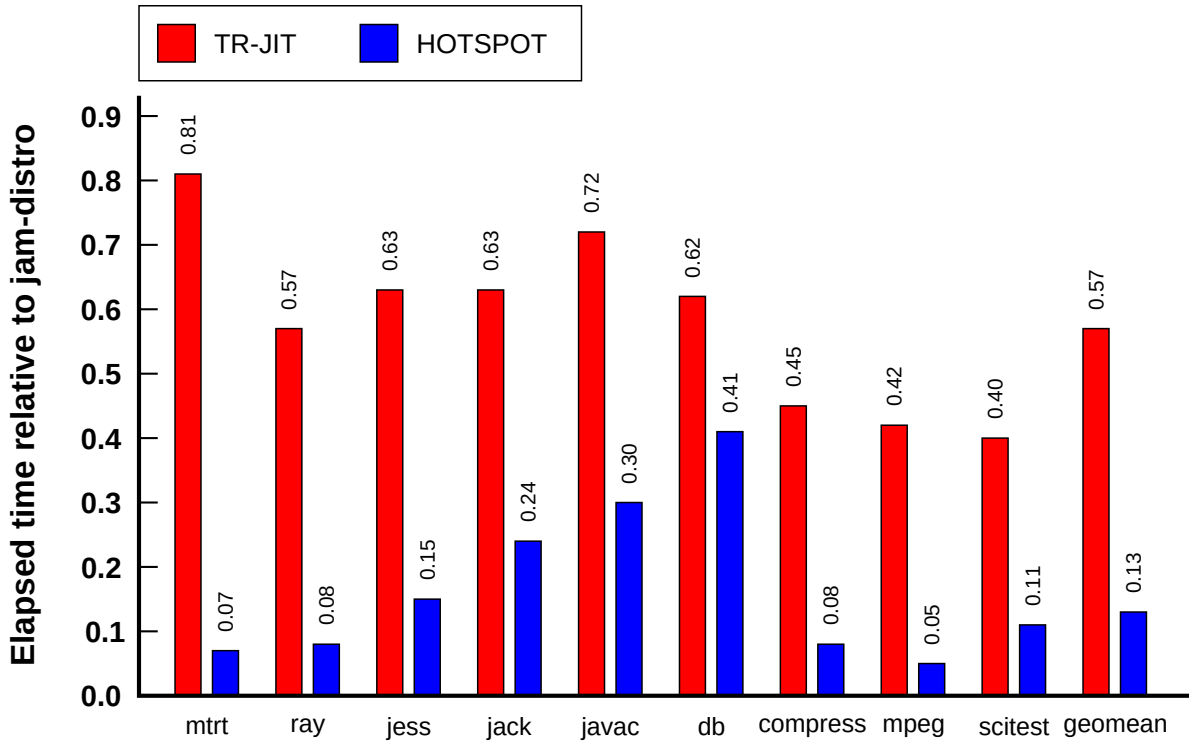



Figure 7.8: Elapsed time performance of Yeti with JIT compared to Sun Java 1.05.0_6_64 relative to JamVM-1.3.3 (direct threading) running SPECjvm98 benchmarks.

Hence, even though it is a floating-point benchmark, it is greatly improved by devirtualizing and inlining these accessor methods.

Figure 7.8 compares the performance of TR-JIT to Sun Microsystems' Java HotSpot JIT. Our current JIT runs the SPECjvm98 benchmarks 4.3 times slower than HotSpot. Results range from 1.5 times slower for `db`, to 12 times slower for `mtrt`. Not surprisingly, we do worse on floating-point intensive benchmarks since we do not yet compile the float bytecodes.

## 7.4 Early Pentium Results

As illustrated earlier, in Figure 3.4, the Intel's Pentium architecture takes a different approach to indirect branches and calls than does the PowerPC. On the PowerPC, we have shown that the two-part indirect call used in Yeti's dispatch loops performs well. However, the Pentium relies on its BTB to predict the destination of its indirect call instruction. As we saw in Chapter 5, when the prediction is wrong, many stall cycles may result. Conceivably, on the Pentium, the unpredictability of the dispatch loop indirect call could lead to very poor performance.

Gennady Pekhimenko, a fellow graduate student at the University of Toronto, ported i-TR to the Pentium platform. Figure 7.9 gives the performance of his prototype. The results are roughly comparable to our PowerPC results, though i-TR outperforms direct threading a little less on the Pentium. The average test case ran in 83% of the time taken by direct threading whereas it needed 75% on the PowerPC.

## 7.5 Identification of Stall Cycles

We have shown that Yeti performs well compared to existing interpreter techniques. However, much of our design is motivated by micro-architectural considerations. In this section, we use a new set of tools to measure the stall cycles experienced by Yeti as it runs.

The purpose of this analysis is twofold. First, we would like to confirm that we understand

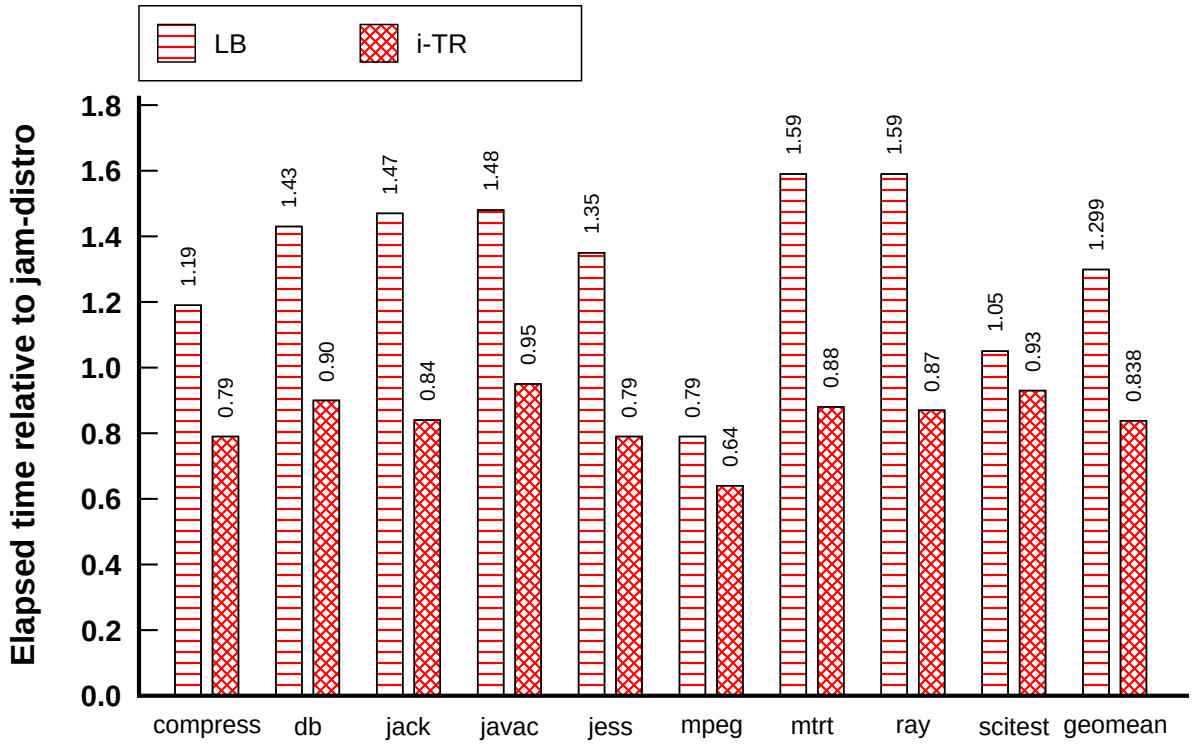



Figure 7.9: Performance of Gennady Pekhimenko's Pentium port relative to unmodified JamVM-1.3.3 (direct-threaded) running the SPECjvm98 benchmarks.

why Yeti performs well. Second, we would like to discover any source of stalls we did not anticipate, and perhaps find some guidance on how we could do better.

### 7.5.1 Identifying Causes of Stall Cycles

Azimi et al [6] describe a system that uses a statistical heuristic to attribute stall cycles in a PowerPC 970 processor. They define a *stall cycle* as a cycle for which there is no instruction that can be completed. Practically speaking, on a PowerPC970, this occurs when the processor's completion queue is empty because instructions are held up, or stalled. Their approach, implemented for a PPC970 processor running K42, a research operating system [18], exploits performance monitoring hardware in the PowerPC that recognizes when the processor's instruction completion queue is empty. Then, the next time an instruction *does* complete they attribute, heuristically and imperfectly, all the intervening stall cycles to the functional unit of the completed instruction. Azimi et al show statistically that their heuristic estimates the true causes of stall cycles well.

The Linux port runs only on a PowerPC 970FX processor[2]. This is slightly different than the PowerPC 970 processor we have been using up to this point. The only acceptable machine we have access to is an Apple Xserve system which was also slightly faster than our machine, running at 2.3 GHz rather than 2.0 GHz.

### 7.5.2 Stall Cycle results

Figure 7.10 shows the results of the Azimi et al's tools to break down stall cycles for various runs of the SPECjvm98 benchmarks.

Five bars appear for each benchmark. From the left to the right, the stacked bars represent

[2]We suspect that the actual requirement is the interrupt controller that Apple packages in newer systems.

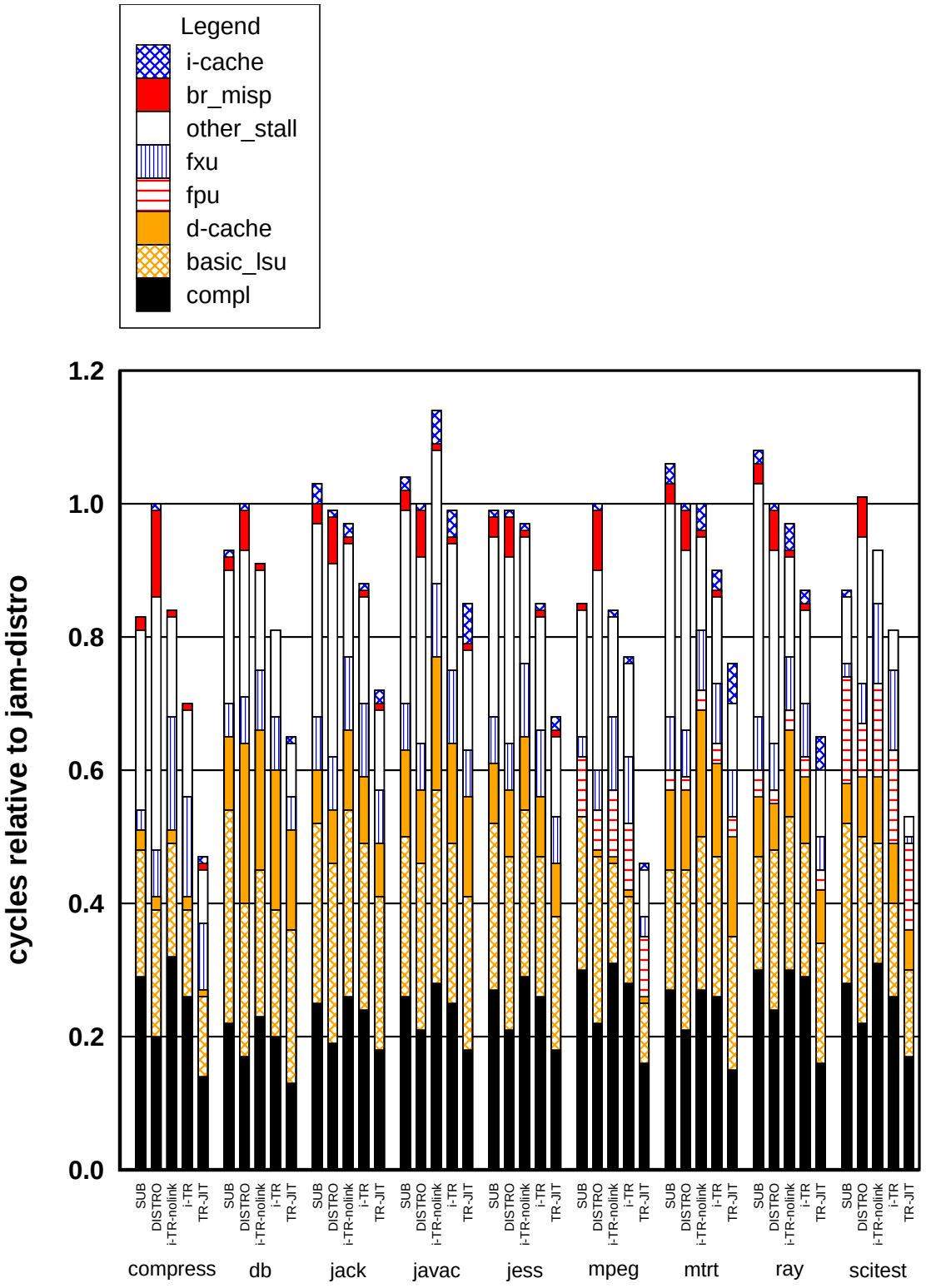



Figure 7.10: Cycles relative to JamVM-1.3.3 (direct threading) running SPECjvm98 benchmarks.

| category name | Description |
|---|---|
| i-cache | Instruction cache misses |
| br_misp | Branch mispredictions |
| compl | Completed instructions. (Cycles in which an instruction did complete) |
| other_stall | Miscellaneous stalls |
| fxu | Fixed point execution unit |
| fpu | Floating point execution unit |
| d-cache | Data cache |
| basic_lsu | Basic load and store unit stalls |

Table 7.3: GPUL categories

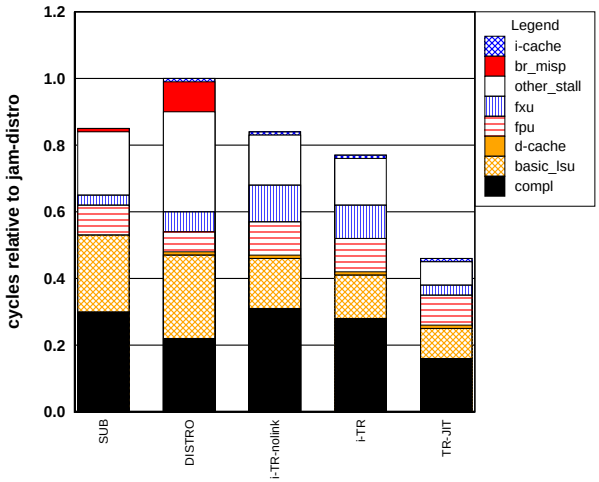
subroutine-threaded JamVM 1.1.3 (SUB) , JamVM 1.1.3 (direct-threaded as distributed, hence DISTRO) and three configurations of Yeti, i-TR-no-link, i-TR and TR-JIT. The y axis, like many of our performance graphs, reports performance relative to JamVM. The height of the DISTRO bar is thus 1.0 by definition. Figure 7.11 reports the same data as Figure 7.10, but, in order to facilitate pointing out specific trends, zooms in on four specific benchmarks.

Each histogram column is split vertically into a stack of bars which illustrates how executed cycles break down by category. Only cycles listed as “compl” represent cycles in which an instruction completed. All the other categories represent stalls, or cycles in which the processor was unable to complete an instruction. The “other_stall” category represents stalls to which the tool was not able to attribute a cause. Unfortunately, the other_stall category includes a source of stalls that is important to our discussion, namely the stalls caused by data dependency between the two instructions of the PowerPC architectures’ two-part indirect branch mechanism[3]. See Figure 3.4 for an illustration of two-part branches.

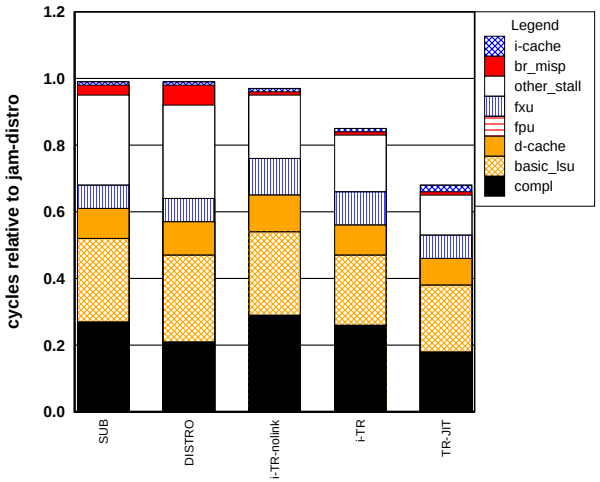
The total cycles executed by each benchmark do not correlate perfectly with the elapsed time measurements reported earlier in this chapter.

For instance, in Figure 7.4, i-TR runs scitest in 60% of the time of direct threading, whereas in Figure 7.11(c) it takes 80%. There are a few important differences between the runs, namely

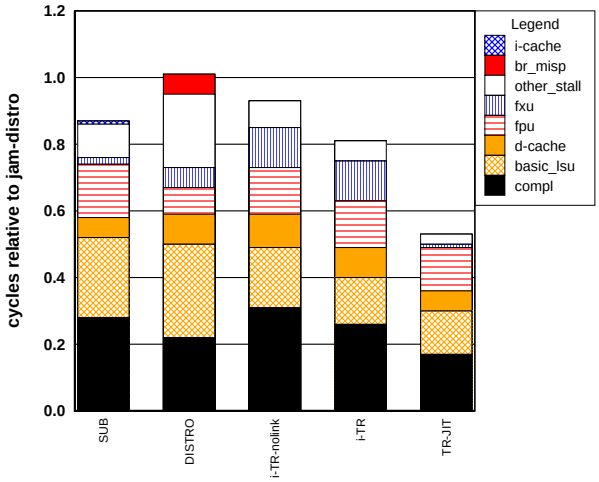
[3]In earlier models of the PowerPC, for instance the 7410, these cycles were called “LR/CTR stall cycles”, as reported by Figure 5.1(b)



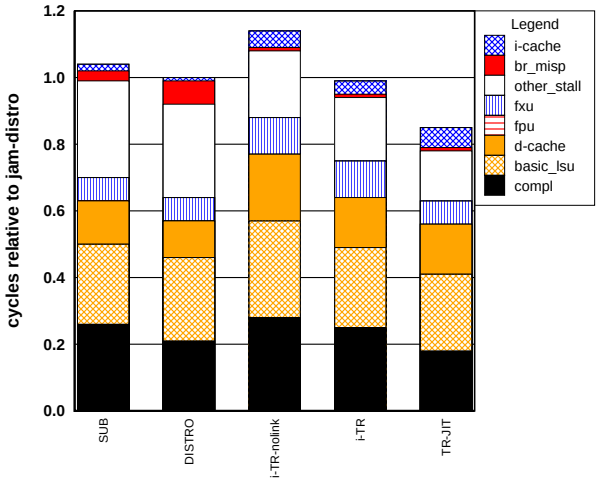

(mpeg) – long int blocks




(jess) – short blocks




(scitest) – long float blocks




(javac ) – trace cache bloat

Figure 7.11: Stall breakdown for SPECjvm98 benchmarks relative to JamVM-1.3.3 (direct threading).

the differences between the PowerPC 970FX and PowerPC 970, the different clock speed (2.3 GHz vs 2.0 GHz) and differences between Linux (with Azimi et al's modifications) and OSX 10.4. We use the data qualitatively to characterize pipeline hazards and not to measure absolute performance.

### 7.5.3 Trends

Several interesting trends emerge from our examination of the cycle reports.

1. Interpreted traces reduce branch mispredictions caused by virtual branch instructions.
2. Simple code we generated for interpreted trace exits stresses the fixed-point execution unit (fxu)
3. Our JIT (TR-JIT) does little to reduce lsu stalls, which is a surprise since many loads and stores to the expression stack are eliminated by the register allocator.
4. As we reduce pipeline hazards caused by dispatch new kinds of stalls arise.
5. Trace bloat, like we observed for javac, can lead to significant stalls due to instruction cache misses.

Each of these issues will be discussed in turn.

#### Branch misprediction

In Figure 7.11(mpeg) we see how our techniques affect `mpeg`, which has a few very hot, very long basic blocks. The blocks contain many duplicate virtual instructions. Hence, direct threading encounters difficulty due to the context problem, as discussed in Section 3.5. (This is plainly evident in the solid red br_misp stack on the DISTRO bar on all four sub figures.)

SUB reduces the mispredictions that occur running `mpeg` significantly – presumably the ones caused by linear regions. Yeti's i-TR technique effectively eliminates the branch mis-