Dynamically Accelerating Client-side Web Applications through Decoupled Execution

Mojtaba Mehrara
University of Michigan, Ann Arbor
mehrara@umich.edu

Scott Mahlke
University of Michigan, Ann Arbor
mahlke@umich.edu

***Abstract*—**

The emergence and wide adoption of web applications have moved the client-side component, often written in JavaScript, to the forefront of computing on the web. Web application developers try to move more computation to the client side to avoid unnecessary network traffic and make the applications more responsive. Therefore, JavaScript applications are becoming larger and more computation intensive. Trace-based just-in-time compilation have been proposed to address the performance bottleneck in these applications. In this paper, we exploit the extra processing power in multicore systems to further improve the performance of trace-based execution of JavaScript programs. In trace-based engines, a considerable portion of execution time is spent on running *guards* which are operations inserted in the native code to check if the properties assumed by the compiled code actually hold during execution. We introduce ParaGuard to off-load these *guards* to another thread, while speculatively executing the main trace. In a manner similar to what happens in current trace-based JITs, if a check fails, ParaGuard aborts the native trace execution and reverts back to interpreting the JavaScript bytecode. We also propose several optimizations including guard branch aggregation and profile-based snapshot elimination to further improve the performance of our technique. We show that ParaGuard can achieve an average of 15% performance improvement over current trace-based compilers using an extra processor on commodity multicore processors.

I. Introduction

JavaScript has become ubiquitous for client side web programming due to its flexibility, ease of prototyping, and portability. Dynamically downloaded JavaScript programs combine a rich and responsive client-side experience with centralized access to shared data and services provided by data centers. The uses of JavaScript range from simple scripts utilized for creating menus on a web page to sophisticated applications that consist of many thousands of lines of code executing in the user's browser. Some of the most visible applications, such as Gmail and Facebook, enjoy widespread use by millions of users. Other applications, such as image processing applications and games, are also becoming more commonplace due to the ease of software distribution.

As JavaScript applications become popular and their complexity grows, the need for higher performance will become essential. However, this is a difficult challenge for dynamically typed languages such as JavaScript. The types of variables and expressions may vary at run-time, thus the compiler must emit generic code that can handle all potential type combinations. This code is then executed through interpretation, which is often extremely slow in comparison to the code generated for statically typed languages such as C or C++.

There is disagreement in the community about the forms of JavaScript applications that will dominate and thus the best strategy for optimizing performance. JSMeter [28] characterizes the behavior of JavaScript applications from commercial websites and argues that long-running loops and functions with many repeated instructions are uncommon. Rather, they are mostly event-driven with thousands of events being handled based on user interactions.

While this characterization of interaction-intensive applications reflects the current dominance of applications such as Gmail and Facebook, it may not reflect the future. More recently, Richards et al. [30] performed similar analyses on a fairly large number of commercial websites and concluded that in many websites, execution time is, in fact, dominated by hot loops, but less so than Java and C/C++. Furthermore, an emerging class of online games and client-side image editing applications are becoming more and more popular. There are already successful examples of image editing applications written in ActionScript for Adobe Flash [8], [10]. There are also many efforts in developing online games and gaming engines in JavaScript [1], [14]. These *compute-intensive* applications are dominated by frequently executed loops and functions.

The main obstacle preventing wider adoption of JavaScript for compute-intensive applications is historical performance deficiencies. These applications must be distributed as native binaries because consumers would not accept excessively poor performance. A circular dependence has developed where poor performance discourages developers from using JavaScript for compute-intensive applications, but there is little need to improve JavaScript performance because it is not used for heavy computation.

This circular dependence is being broken through the development of new dynamic compilers for JavaScript. TraceMonkey, a trace-based JavaScript engine, was developed for the Firefox web browser to remove some of the inefficiencies associated with dynamic typing [19]. TraceMonkey identifies hot bytecode sequences and compiles them to native machine code with statically assumed types. As long as the sequences (traces) remain type-stable, execution remains in the type-specialized machine code. TraceMonkey works at the granularity of individual loops, and therefore, is very well suited

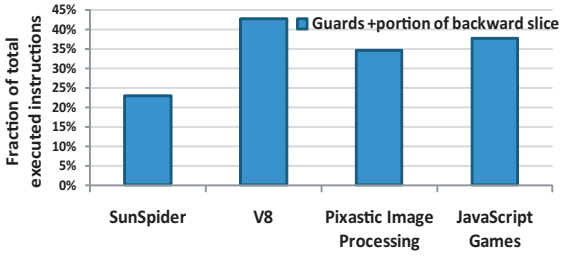


Fig. 1. Fraction of instructions devoted to computing guards across four groups of benchmarks: SunSpider, V8, Pixastic image processing applications, and a set of JavaScript games. These bars include guards and portion of the backward slice only needed by guards and not used elsewhere.

for compute-intensive web applications.

While compiling hot traces to the native code, TraceMonkey inserts runtime checks, called *guard* instructions, into the trace to check for type, control flow, and other assumptions that were made during the JIT compilation process. These checks are heavily biased not to fire as the vast majority of the time the types do not vary and a single control flow path is dominant [19]. However, these guards comprise a significant fraction of total executed instructions. Figure 1 presents the overhead of guards consisting of the guard instructions themselves as well as the dependent computation used by the them. These are the instructions only used by guards and are not needed elsewhere in the trace. The average overhead is presented for four groups of applications: SunSpider [13] and V8 [15] benchmark suites, and two sets of applications from the image processing and gaming domains (more details on the benchmarks are provided in Section V). These values range from a low of 22% to a high of 42%, which represents a significant runtime penalty.

In this work, we focus on reducing this overhead using a multi-threaded dynamically decoupled execution framework called *ParaGuard*. We decompose traces generated by TraceMonkey into two concurrent threads. The main thread consists of the code to implement the bulk of the user program, while the *ParaGuard* thread performs most of the runtime checks. With this model, the main thread speculatively executes ahead assuming that the checks will not fire and the common execution scenario will proceed. When a check does fail, it reverts back to the interpreter and safely discards the improper speculative work. During speculative execution, the program is sandboxed to make sure no catastrophic execution failures happen until ParaGuard checks have been validated. In multicore systems with under-utilized cores, we can execute the main and guard threads concurrently to increase performance.

The contributions offered by this paper are as follows:

- We propose ParaGuard, a method to dynamically decompose a type-specialized trace into two concurrent threads: the first speculatively performs the core computation along the expected path of control and the second verifies that the assumptions used to create the trace are valid.
- We introduce several optimizations including guard branch aggregation and profile-based snapshot elimination to increase the efficiency of the decoupled execution.
- We evaluate the ParaGuard system on a set of JavaScript applications from the gaming and image processing domains, in addition to two popular benchmark suites, SunSpider and V8.

II. Background

In statically typed languages such as C or C++, the compiler can generate efficient machine code based on the type information provided by the programmer. However, in dynamically typed languages such as JavaScript, variable types can change at runtime and therefore, the compiler cannot generate machine code specialized for only one specific type. This forces the compiler to generate generalized machine code with the ability to handle potential dynamic type changes, causing the code to be considerably slower than the statically typed machine code. Some static compile-time type inference techniques can be applied to dynamically typed languages, but such techniques are far too slow for a language like JavaScript that needs to be loaded and compiled quickly in the web browser.

There have been a number of efforts to efficiently compile and execute JavaScript applications on different browsers. One of the most recent proposals is TraceMonkey [19] by Mozilla which is implemented on top of SpiderMonkey [12] and is now integrated in their web browser, Firefox [7].

TraceMonkey uses a trace-based compilation method that reduces JavaScript execution time by exploiting high performance type-specialized machine code when possible. It starts off by running the JavaScript application in a bytecode interpreter and at the same time identifies and records hot bytecode execution sequences. These sequences, called traces, are then compiled to native code. In TraceMonkey, traces are formed out of individual hot loops. This choice is based on the assumption that hot loops are mostly type-stable, thereby allowing most of the program execution to be expressed by type-specialized and natively compiled traces.

Each compiled trace consists of a single path in the program with a specific value-type mapping. However, this type-mapping is not guaranteed to be always correct, because different code paths may be taken or different types may be assigned to a value in subsequent loop iterations. Therefore, executing the same trace for later loop iterations is based on the speculation that the path and types will match what was observed during recording. These speculations are verified using a number of checks (called *guards*) along the trace. The guards are inserted wherever there is a need to check for alternate typing, control flow paths or other runtime checks (as described in the beginning of Section III). If these checks fail, the trace exits and reverts back to interpreting the bytecode. Likewise, if the exit becomes hot, a branch trace is generated and compiled to cover the new path. In this way, a trace tree is eventually formed which covers all hot paths in the loop.

Figure 2 describes the major phases of JavaScript execution in TraceMonkey. These phases happen in the *trace monitor* which coordinates the whole tracing process. Initially, the program starts in the bytecode interpreter, and when the interpreter reaches a loop edge, the trace monitor is called

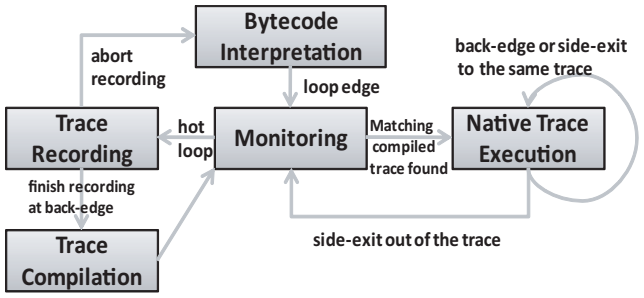


Fig. 2. JavaScript tracing and type specialization in TraceMonkey. This state machine describes how the trace monitor manages trace-based just-in-time compilation.

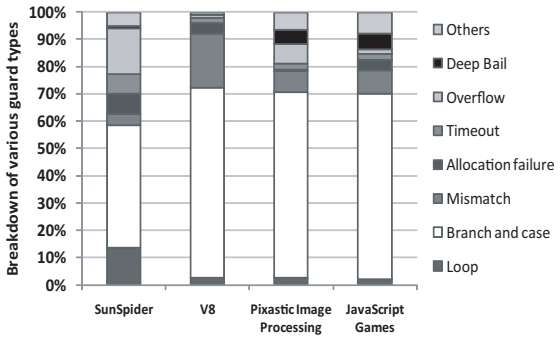


Fig. 3. Breakdown of different types of guards in SunSpider, V8, Pixastic image processing and JavaScript games.

to determine whether a new trace should be recorded or an existing native trace could be executed for the loop. At the start of execution, since there are no compiled traces, the trace monitor simply profiles the number of loop edge crossings and enters the recording state after a loop becomes hot. During recording, the code along the trace is recorded in a low-level intermediate representation (LIR) which encodes all the operations and types in the trace. The LIR also contains guards to ensure that the control flow and types are identical to what was observed during recording. If the recorder is unable to continue recording, for example when faced with `eval` calls or reaching the trace length limits in a small-memory device, it chooses to abort the recording. On such an abort, the monitor discards the recorder and returns to the monitoring state. The monitor also keeps track of how many times the recording has failed for a trace starting at each program counter (PC) value. Therefore, if a particular PC causes too many aborted recordings, the monitor *blacklists* the PC and will not attempt to record it again.

The recording is finished when execution reaches the loop header or exits the loop. Subsequently, the trace is compiled to the native code based on the types and control path of the recorded trace. From then on, whenever the monitor interprets a backward jump to a PC with a matching compiled trace (with the same type map), it enters native execution mode. In this mode, before calling the native trace, the monitor allocates a trace activation record containing imported local and global variables, temporary stack space, and space for arguments to native calls. The monitor then calls the trace native code with the activation record as an argument. The native code returns with a pointer to a structure containing information about how the trace exited. Based on this information, the monitor restores interpreter state by copying back the imported variables from the trace activation record.

The monitor behaves differently afterwards, based on the success of the trace return. If the trace exits unsuccessfully (e.g., due to having garbage collection triggered, running out of native stack, or noticing other abnormal conditions), the monitor returns to the monitoring state. However, if the trace exits successfully (e.g., due to running out of native code or hitting a branch condition for which no native code exists yet), the monitor checks whether the side exit PC has become hot or not. If not, it just keeps monitoring the interpretation to find other hot traces. If it has become hot, the monitor moves on to the recording state immediately, starting a new branch trace from that point and patching the side exit to jump directly to that branch. Using this approach, a single trace expands to a multiple-exit trace which could span a fairly large portion of the frequent execution graph.

In practice, loops are typically entered with only a few different combinations of variable types. Therefore, a small number of traces per loop is sufficient to run a program efficiently. TraceMonkey is able to achieve speedups of 2x to 20x on programs for which tracing is feasible [19].

III. ParaGuard: Concurrent Guard Execution

During LIR generation, the following categories of guards can be inserted into the trace.

Loop guards: They are inserted at the end of the loop and check for the loop termination condition.

Branch and case guards: When the LIR corresponding to a trace is generated, conditional branches and case statements are first replaced with unconditional ones, taking the same path that had been taken during trace recording. Guard instructions are then inserted to actually check the branch/case conditions and abort the trace if a different path needs to be taken.

Condition mismatch guards: These guards are inserted to terminate trace execution in case a condition, relied upon at recording time, no longer holds. In some of these situations, the alternate path of execution is so rare or difficult to handle in the native code, that it is preferable to have it interpreted rather than traced and compiled. One example is a negative array index access which requires string-based property lookups, compared to a positive index access which is merely a simple memory access. Type mismatch guards are also included in this category, and they check if the actual type during native execution matches with what was observed during recording.

Miscellaneous guards: There are several other categories of guards such as allocation failure, execution timeout, variable overflow, and deep bail guards. Deep bail guards are triggered when during the execution of a native C function call in the trace, a trace exit is triggered.

Figure 3 shows the average relative ratio of different guard types in SunSpider and V8 suites, and our suite of image

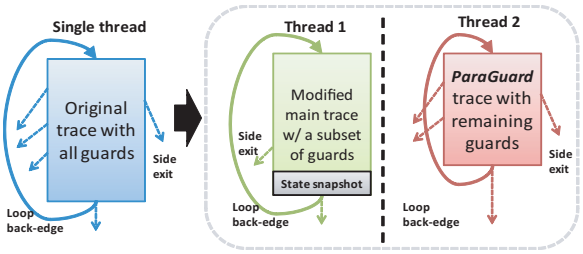


Fig. 4. Offloading guard execution to the ParaGuard thread.

processing programs and JavaScript games. Miscellaneous guards comprise the top five sections in each bar. As can be seen, branch guards are the most frequently generated guards across all benchmarks. Condition mismatch, loop and overflow guards are other common ones.

In the *ParaGuard* technique (Figure 4), the majority of guards are moved to another trace (ParaGuard trace) and are executed in a separate thread (ParaGuard thread), in parallel to the main trace. ParaGuard trace code is generated along with the main trace and is invoked at the same time during trace monitoring. The following subsections describe how we generate ParaGuards and restore the correct state of the interpreter after a ParaGuard is triggered and the trace is aborted. In Section IV, two optimizations are introduced to further improve the performance of our technique.

A. ParaGuard Generation

The optimizations in TraceMonkey are performed in two pipelined phases over the trace. During trace recording, immediately after the recorder emits an LIR instruction, the instruction is sent through the forward optimization pipeline. This forward pass consists of several optimizations including common subexpression elimination and expression simplifications such as constant folding. The second phase is a backward pass which goes through the whole trace from bottom to top after trace recording is complete. The optimizations in this pass include dead code elimination and dead data-stack and call-stack store elimination. After an LIR instruction passes the last stage in the optimization pipeline, the code generator emits the corresponding machine instructions.

Traditional guards are generated and inserted in the LIR during the forward pass. However, since we want to move guard instructions along with the LIR instructions that they depend on (their backward slice), we need to generate ParaGuards as an extra pipeline stage after all optimizations in the backward pass. We call this stage, *guard promotion*. The goal of guard promotion is to identify LIR instructions (guards and non-guards) that can be moved to the ParaGuard trace. A non-guard instruction is moved to the ParaGuard trace if it is only used for computing the inputs of a relocated guard. Furthermore, some instructions are marked for duplication in the ParaGuard trace, since they need to be re-executed there to minimize communication between the ParaGuard and main threads. During guard promotion, two groups of instructions are constructed. The first category is "to-be-copied" which contains the instructions duplicated on both the main and

```
var myArray = new Array();
function init() {
  var j = 0;
  for (j = 0; j < 200; ++j)
    myArray[j] = j*2;
}
```

Fig. 5. Sample JavaScript source code.

ParaGuard traces. The second group, called "to-be-moved", consists of all instructions that are moved from the main trace to the ParaGuard trace by the end of the guard promotion pass. This pass is performed in two steps:

Step 1: This is essentially a partial implementation of backward slicing. Starting from each guard instruction in the trace, the compiler keeps track of *def* instructions for the guard's source operands. Likewise, it tracks *def*s of the source operands of those *def* instructions. This procedure is continued recursively, traversing *def/use* chains and marking *def*s as "to-be-copied". The destinations of these *def* instructions are also kept in a list for use in the second step. To avoid violating memory consistency between the main and ParaGuard thread, tracking *def*s is stopped after reaching a load instruction. Because if the load is copied or moved to the ParaGuard trace, the code needs to ensure that the load in the ParaGuard thread is not executed before the corresponding store in the main thread. Enforcing this requires adding locking primitives, which can cause high overheads.

Step 2: The goal of this step is to remove the *def*s that are only used in the guard's backward slice from the main trace. First the candidate guard for moving is marked as "to-be-moved". As the trace is traversed backwards, all *use*s of the candidate guard's source operands are recursively kept in a "use-set". When a *def* marked as "to-be-copied" (during step 1) is reached, its "use-set" is checked to see whether all its members are marked as "to-be-moved". If so, it is clear that this *def* is not going to be used in the main trace before the guard instruction, if all "to-be-moved" instructions are moved to the ParaGuard trace. Furthermore, a *def*'s destination liveness after the guard instruction should also be checked. In order to do that, the live set at the guard instruction is used and if the *def*'s destination operand is not a member of this live set, the *def*'s category can safely be changed from "to-be-copied" to "to-be-moved". These live sets are already generated prior to the guard promotion pass. To summarize, a *def* must meet three conditions to qualify for relocation to the ParaGuard trace:

1. It is marked as "to-be-copied".
2. All its uses before the guard are marked as "to-be-moved".
3. Its destination is not live after the guard instruction.

In addition to this analysis, guard promotion uses a heuristic that rejects promotion of the guard instructions whose backward slice is either very small or should be mostly copied to the ParaGuard trace rather than moved. Therefore, by the end of the guard promotion pass, some guards still remain in the main trace.

At runtime, live-in values to the ParaGuard trace are copied to a per-guard single-reader/single writer buffer, similar to the

```
label1:
(*)1 : cx = ldq state[16]          // load context pointer
   2 : ld1 = ld sp[-8]             // load 'j' from stack
(+)3 : ld2 = ld cx[0]              // load context object
(+)4 : eq3 = eq ld2, 0             // check if context is valid
(+)5 : xf eq3                      // side exit if it's not
(*)6 : $globl0 =ldq state[848]     // load myArray pointer from
                                   // trace activation record
   7 : stqi sp[0] = $globl0        // store myArray on stack
   8 : sti sp[8] = ld1             // store j on the stack
   9 : sti sp[24] = 2              // store 2 on the stack
(*)10: mul1 = mul ld1, 2           // multiply j by 2
(+)11: ov2 = ov mul1               // check overflow on mul op
(+)12: xt4: xt ov2                 // side exit if mul overflows
(+)13: eq2 = eq mul1, 0            // check if mul1 is zero
(+)14: xt eq2                      // side exit if so
   15: sti sp[16] = mul1           // store mul result on stack
(*)16: ldq1 = ldq $globl0[8]       // load myArray class
(+)17: qi1 = qiand ldq1,           //
     quad #FFFFFFFF:FFFFFFFC
(+)18: cl = quad #0:803D20         //
(+)19: arrayg = qeq qi1, cl        // check if class is an array
(+)20: xf arrayg                   // side exit if not
   21: returng=js_Array_set(       // set myArray element
     $globl0 ld1 mul1)
   22: eq1 = eq returng, 0         // check js_Array_set return value
   23: xt eq1                      // side exit if failed
(*)24: add1 = add ld1, 1           // add 1 to j
(+)25: ov1 = ov add1               // check add for overflow
(+)26: xt ov1                      // side exit if overflows
   27: sti sp[-8] = add1           // store add result on stack
   28: sti sp[8] = 200             // store 200 on stack
(*)29: lt1 = lt add1, 200          // check loop condition
(*)30: xf lt1                      // exit trace if finished
   31: sti sp[-8] = add1           // store add result on stack
(*)32: j -> label1                 // jump back to the top
```

Fig. 6. Original TraceMonkey's Low-level IR for the source code in Figure 5. Instructions marked with (*) are to be copied and the ones with (+) are to be moved to the ParaGuard trace.

```
label1:
1  : cx = ldq state[16]            // (*) load context pointer
2  : ld1 = ld sp[-8]               // load ''j'' from stack
PG1: st shared_buf[0] = ld1        // store ld1 in the shared_buff
6  : $globl0 =ldq state[848]       // (*) load myArray pointer
                                   // from trace activation record
7  : stqi sp[0] = $globl0          // store myArray on stack
8  : sti sp[8] = ld1               // store j on the stack
9  : sti sp[24] = 2                // store 2 on the stack
10 : mul1 = mul ld1, 2             // (*) multiply j by 2
15 : sti sp[16] = mul1             // store mul result on stack
16 : ldq1 = ldq $globl0[8]         // (*) load myArray class
21 : returng=js_Array_set(         // set myArray element
    $globl0 ld1 mul1)
22 : eq1 = eq returng, 0           // check js_Array_set return val
23 : xt eq1                        // side exit if failed
24 : add1 = add ld1, 1             // (*) add 1 to j
PG2: count = add count, 1          // inc snapshot counter
27 : sti sp[-8] = add1             // store add result on stack
28 : sti sp[8] = 200               // store 200 on stack
29 : lt1 = lt add1, 200            // (*) check loop condition
30 : xf lt1                        // (*) exit trace if finished
PG3: eq2 = eq count, N             // check snapshot condition
PG4: jt eq2 -> label2              // jump if snapshot needed
31 : sti sp[-8] = add1             // store add result on stack
32 : j -> label1                   // (*) jump back to the top
label2:
   barrier paraguard_finish
   take_snapshot()
   count = 0
   j -> label1
```

(a) Main Trace LIR

```
label1:
1' : cx = ldq state[16]            // (*) load context pointer
3  : ld2 = ld cx[0]                // (+) load context object
4  : eq3 = eq ld2, 0               // (+) check if context is valid
5  : xf eq3                        // (+) side exit if it's not
6  : $globl0 =ldq state[848]       // (*) load myArray pointer
PG5: barrier shared_buf[0]         // wait for of shared_buf[0]
PG6: ld1 = ld shared_buf[0]        // load ld1 from the shared_buf[0]
10' : mul1 = mul ld1, 2            // (*) multiply j by 2
11 : ov2 = ov mul1                 // (+) check overflow on mul op
12 : xt4: xt ov2                   // (+) side exit if mul overflows
13 : eq2 = eq mul1, 0              // (+) check if mul1 is zero
14 : xt eq2                        // (+) side exit if so
16': ldq1 = ldq $globl0[8]         // (*) load myArray class
17 : qi1 = qiand ldq1,             // (+)
   quad #FFFFFFFF:FFFFFFFC
18 : cl = quad #0:803D20           // (+)
19 : arrayg = qeq qi1, cl          // (+) check if class is an array
20 : xf arrayg                     // (+) side exit if not
PG7: count = add count, 1          // inc snapshot counter
24': add1 = add ld1, 1             // (*) add 1 to j
25 : ov1 = ov add1                 // (+) check add for overflow
26 : xt ov1                        // (+) side exit if overflows
29': lt1 = lt add1, 200            // (*) check loop condition
30': xf lt1                        // (*) exit trace if finished
PG8: eq3 = eq count, N             // check snapshot condition
PG9: jt eq3 -> label2              // jump if snapshot needed
32': j -> label1                   // (*) jump back to the top
label2:
   bdcast paraguard_finish
   j -> label1
```

(b) ParaGuard Trace LIR

Fig. 7. Main and ParaGuard traces after the guard promotion pass.

buffers in [27], which is written by the main trace and read by the ParaGuard trace. Initializing these per-guard buffers is done in the ParaGuard thread and is off the critical path in the main trace. The initial sizes of these buffers are determined at compilation time and in case more space is needed at runtime, they are dynamically expanded. During native execution, ParaGuard trace can start or resume execution once these values are written in the buffers by the main trace.

Figure 5 shows an example JavaScript code snippet. TraceMonkey's LIR for this code can be seen in Figure 6. Backward slices for each guard are highlighted with a different gray shade. Instructions belonging to multiple backward slices are highlighted with the same shade as the earliest observed guard in the trace. For instance, the backward slice for guard instruction 30 consists of instructions 29, 24 and 2. Likewise, the backward slice for instruction 26 are instructions 25, 24 and 2, and for instruction 23 are instructions 22, 21, 10, 6 and 2. Instructions marked with (*) are "to-be-copied" and the ones with (+) are "to-be-moved" after performing the guard promotion algorithm on the guards. This algorithm decided not to move the guard at instruction number 23, since it would have only saved two instructions (22 and 23) on the main trace, while either `js_Array_set` had to be re-executed in the ParaGuard trace or its return value had to be copied to the ParaGuard trace buffer.

Finally, Figure 7 shows the modified main trace along with the generated ParaGuard trace after applying guard promotion. The same gray shades have been applied to guard instructions' backward slices. `PG*` instructions highlighted in black are added to these traces during guard promotion. `PG1` copies `ld1` to the shared buffer between the main and ParaGuard traces. `PG5` is the barrier waiting for this value in the ParaGuard trace and `PG6` is loading it from the shared buffer. As the figure illustrates, guard promotion has moved 13 out of 32 instructions in the original trace, while only adding four instructions. Instructions `PG2`, `PG3`, `PG4`, `PG7`, `PG8` and `PG9`

are used for taking the native state snapshot for interpreter state recovery as described in the next subsection.

B. Recovering Interpreter State using Selective Snapshots

As mentioned in Section II, before invoking a trace, the interpreter builds a trace activation record that consists of the temporary stack space, space for arguments to native calls, and all imported global and local variables. These global and local values are copied from i the interpreter state to the trace activation record and the trace is later called like a normal call-through-pointer in C. After a guard is triggered and the trace call returns, the interpreter state is restored by copying the imported global and local variables from the trace activation record back to the interpreter state.

When using ParaGuard, this process gets more complicated. Since the guards trigger asynchronously, the main thread may have corrupted its state by executing instructions past the original guard location and overwriting the correct state. Therefore, some form of checkpointing support is needed for imported native variables, so that when a guard triggers in the ParaGuard trace, execution can roll back to a previous snapshot of the correct execution state.

Traditional rollback support such as those in software transactional memory would incur a high performance overhead and is unacceptable here. Thus, instead of making a backup copy of memory locations on every memory write, we use a *bulk* snapshot mechanism in which the frequency of taking memory snapshots is reduced to every *N* iterations. The exact value of *N* is determined dynamically according to a runtime heuristic which is based on the loop's instruction count, total iteration count, and number of memory operations per iteration. When the execution on the main trace reaches the loop guard and the trip count is a multiple of *N*, it stops at a barrier, waiting for the ParaGuard thread to catch up. In most cases, there is no waiting, because the ParaGuard trace is shorter than the main trace. Subsequently, the main trace takes the state snapshot, after which it continues executing. Since TraceMonkey does not perform tracing if the code path contains I/O accesses, the snapshot taking mechanism does not have to deal with checkpointing I/O operations.

In order to further reduce the overhead of bulk snapshots, a *selective* snapshot is taken which only includes critical memory locations. These locations are all trace live-outs including stack, heap and global variables, objects and data structures. Snapshots of scalar non-object variables are taken by simply cloning their value, while live-out objects are deep-copied. The deep-copying process is set up such that there are no duplicate copies of the same object in the snapshot in case of cycles in the object graph or when two variables point to the same object. For live-out arrays, an accumulative snapshot mechanism is employed where after an array snapshot is taken before the loop, during each N iteration period at runtime, the minimum and maximum accessed array indices are recorded. Subsequently, all elements between these indices are stored into the array's accumulative snapshot. Since all array indices are already passed to the ParaGuard trace to be checked by the condition mismatch guards, keeping track of these maximum and minimum values is performed inside the ParaGuard trace. Therefore, they impose no extra overhead on the main trace. These values are later sent back to the main trace at the time of periodic snapshot taking. Figure 8 shows the extra code for this purpose that needs to be added to Figure 7(b).

```
lt0 = lt ld1,min0   // compare with min index
jt -> updateMin     // if smaller, replace min
gt0= gt ld1,max0    // compare with max index
jt -> updateMax     // if larger, replace max
label0: ...
        ...
updateMin:
    min0 = ld1
    j -> label0
updateMax:
    max0 = ld1
    j -> label0     // resume execution
```

Fig. 8. Extra code added after PG7 in Figure 7(b). label0 is inserted right before instruction 24'. updateMin and updateMax code segments are inserted after the label2 code segment.

TraceMonkey uses a mark-and-sweep garbage collector (GC) and has an API function to add variables to the GC's *root set* to prevent anything the root points to from getting collected. Since there will be no references to the snapshots from within the JavaScript application, the garbage collector needs to be asked explicitly not to touch them until the next snapshot is taken by adding the snapshot entries to the root set. Furthermore, because heap objects are deep-copied while taking snapshots, no object in the snapshots points back to the actual application heap. Therefore, although as explained later, snapshots are recovered once a GC is triggered, in theory, there would be no issue of the GC collecting objects in the heap that are pointed to by the snapshot.

When a guard triggers inside the ParaGuard or the main trace, the runtime aborts both threads by sending a signal, restores the previous snapshot and moves back to the interpreter. The rollback operation itself does not add extra overhead compared to the original tracing technique, since it performs the same value forwarding that would have been done for updating the interpreter's state using the native trace data.

Another important issue is what happens when a GC is scheduled. In the original tracing technique, the trace aborts when a GC is invoked. In ParaGuard, the latest correct snapshot is restored after a GC call is triggered. The control is later handed off to the interpreter from the execution location of the previous snapshot. Finally, in order to ensure execution safety in the main trace and avoid catastrophic failures such as null pointer dereference in the native code, signal handlers were defined to catch runtime exceptions, roll back execution to a previous snapshot and switch to the interpretation mode.

In Figure 7(a), instructions PG2, PG3 and PG4 are used to branch to label2 every *N* iterations. At label2, the main thread waits on a condition, set by the ParaGuard thread and marks the end of its execution. When the condition is set, the main trace starts to take the snapshot. Likewise, PG7, PG8, and PG9 are used to branch to label2 in the ParaGuard trace. After branching, the ParaGuard thread broadcasts the barrier release condition to the main thread.

IV. Optimizations on ParaGuard

In order to further improve the performance benefit of guard promotion, two additional optimizations are introduced. As mentioned in Section III-B, before starting the snapshot taking process, the main thread needs to wait for the ParaGuard thread to catch up. Therefore, the ParaGuard thread should be made as fast as possible. We introduce the *guard branch aggregation* optimization, during which, mid-trace guard conditions are aggregated into a single variable, branches are removed, and at the end of each *N* iterations, the single condition variable is checked for any possible triggered guard. Furthermore, taking snapshots can impose a high overhead on the runtime. To tackle this issue, we propose *profile-based snapshot elimination*, in which, based on a profile of previous executions, the guards that are likely to trigger are kept on the main trace, and snapshots are removed altogether from the program.

A. Guard Branch Aggregation

Taking a snapshot of the trace state at every N iterations gives us the opportunity to perform another optimization, called guard branch aggregation, in the ParaGuard trace. At the end of each N iteration chunk, we only need to know if trace execution was successful or not and knowing which guard actually triggered is not important. Regardless of the triggered guard, execution is started from the previous snapshot. Therefore, guard branch executions can be postponed until the end of each N iteration execution chunk in the ParaGuard trace. The two final instructions for every guard are the guard condition generator and the branch itself. Guard branch aggregation combines all guard conditions to a single variable which is later checked by a final branch at the end of the trace after each N iteration period. After applying this optimization, we have essentially converted a trace with a single input and multiple output edges, to one with a single input and two output edges. One downside to using this approach is that in case one of the middle guards fails, the trace has to execute until the end of the iteration chunk. However, in type-stable loops this does not cause any serious performance issues.

B. Profile-based Snapshot Elimination

In some traces, the overhead of taking snapshots turns out to be quite high, mainly due to the high overhead of taking heap and array snapshots. In these traces, the number of unique memory updates per loop is high and causes the snapshot taking mechanism to be inefficient. This effect can be detected early on during trace execution by monitoring the snapshot taking overhead. When detected, the native trace is aborted and the execution falls back to the original tracing mode without guard promotion. After switching to normal tracing execution, triggered guards are recorded and stored on the client. Since these operations are done inside the JavaScript engine, the profile information can be stored on the client's file system. During the next execution of the same JavaScript program on the client, the guard promotion phase only moves the guards that, according to the stored profile, have not triggered during previous executions. After guard promotion, since no snapshots are taken, if a guard is triggered in the ParaGuard trace, the execution aborts native execution, reverts back to interpretation from the beginning of the loop and adds that guard to the profile for use during future executions.

However, if a guard is triggered in the main trace, extra measures should be taken to enable the interpreter to continue from the guard point rather than the beginning of the loop. During guard promotion, the main execution thread stores the sequential order of all guards (both in the main and ParaGuard traces) in a list referenced by the program counter. If a guard triggers in the main trace, it checks to see if all previous guards in the ParaGuard trace have passed successfully. If so, it falls back to the interpreter and continues interpretation from the guard point. Otherwise, it waits for the remaining guards in the ParaGuard trace to pass. Meanwhile, if a guard triggers in the ParaGuard trace, the execution rolls back to the beginning of the loop in the interpretation mode.

The ParaGuard execution model when applying profile-based snapshot elimination is that the first time a JavaScript application is executed, profile is collected if taking snapshots seem to be too costly. From then on, whenever the same application is run on the client, this profile information can be used and updated. Therefore, the first execution of the application, in the worst case, is almost as fast as the baseline tracing execution. In later executions, the application will be enjoying the extra performance benefits of ParaGuard.

V. Experimental Evaluation

A. Methodology

We evaluated our technique on the TraceMonkey version distributed with Firefox 3.7a1pre using four sets of benchmarks. In addition to the two popular benchmark suites, SunSpider [13] and Google V8 [2], we put together two other suites consisting of 12 image processing filters and 5 games implemented in JavaScript. The image processing filters were extracted from the Pixastic JavaScript Image Processing Library [11]. This library contains 28 filters and effects, out of which the 12 most compute-intensive filters were selected. In the JavaScript game suite, four of the benchmarks (Collision demo [3], Thunder fighter [4], Super JS fighter [5], and Invaders from earth [6]) are demos written using the gameQuery JavaScript game engine [1]. The last benchmark is a PacMan game written in JavaScript [9]. All benchmarks were run 10 times, and the average execution time is reported.

In evaluating the profile-based snapshot elimination optimization, we used different input sets for profiling and actual execution in all 4 benchmark suites. In SunSpider and V8, default inputs are used for actual execution and smaller inputs were generated for the profile run. For the image processing benchmarks, different images were used for profiling and execution. In the gaming benchmarks, since the input to all of them involved some kind of random element along with interactions with the user, the evaluation was more involved. In order to make the performance comparisons feasible, the fact that the behavior of these programs are uniform during the execution time was exploited. Therefore, they were

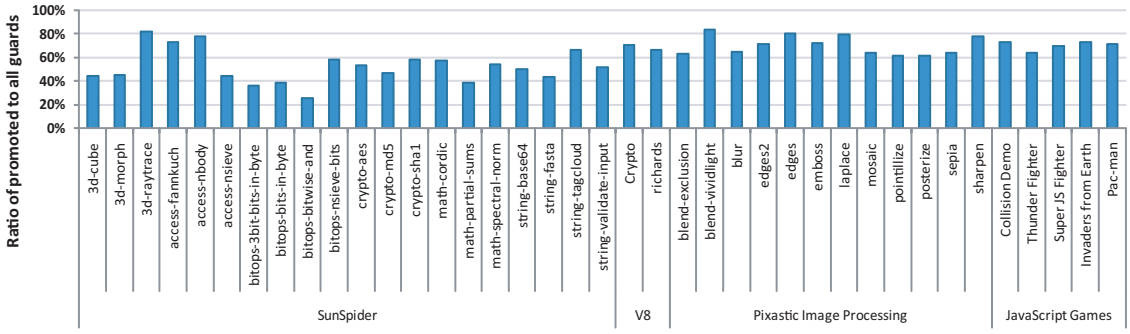


Fig. 9. Ratio of promoted guards to total number of guards.

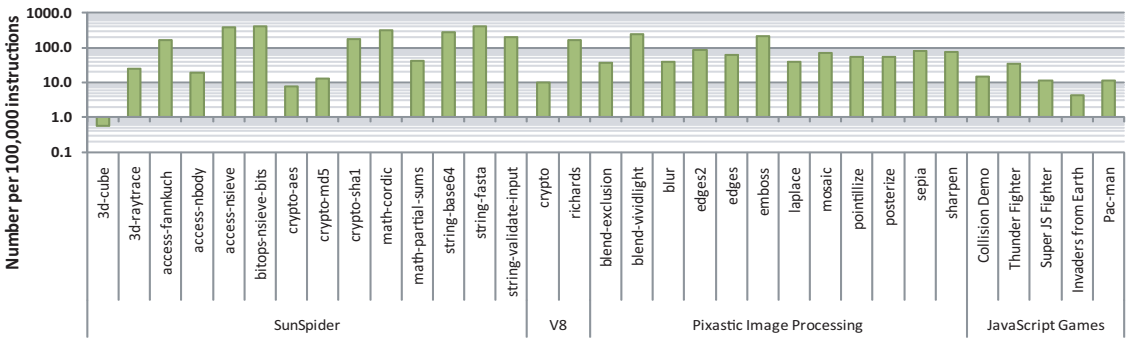


Fig. 10. Number of triggered guards in ParaGuard during every 100,000 instructions in the guard promotion technique without applying the profile-based snapshot elimination. The y-axis is in logarithmic scale.

executed for a fixed number of events at the beginning of the benchmark without any user interaction involved. All random events during the execution were recorded and fed back to the program for all runs (different random events were recorded for profiling and actual runs). For instance, in the `PacMan` game, the paths *ghosts* were taking were fixed and the application ran until a *ghost* hit the *PacMan* which stayed still at its original position. Likewise, in the `Collision Demo` benchmark, all box locations, orientations and movement paths were fixed and the benchmark ran until 10 small boxes collided with the main box in the center. Similar measures were taken in the other three programs as well.

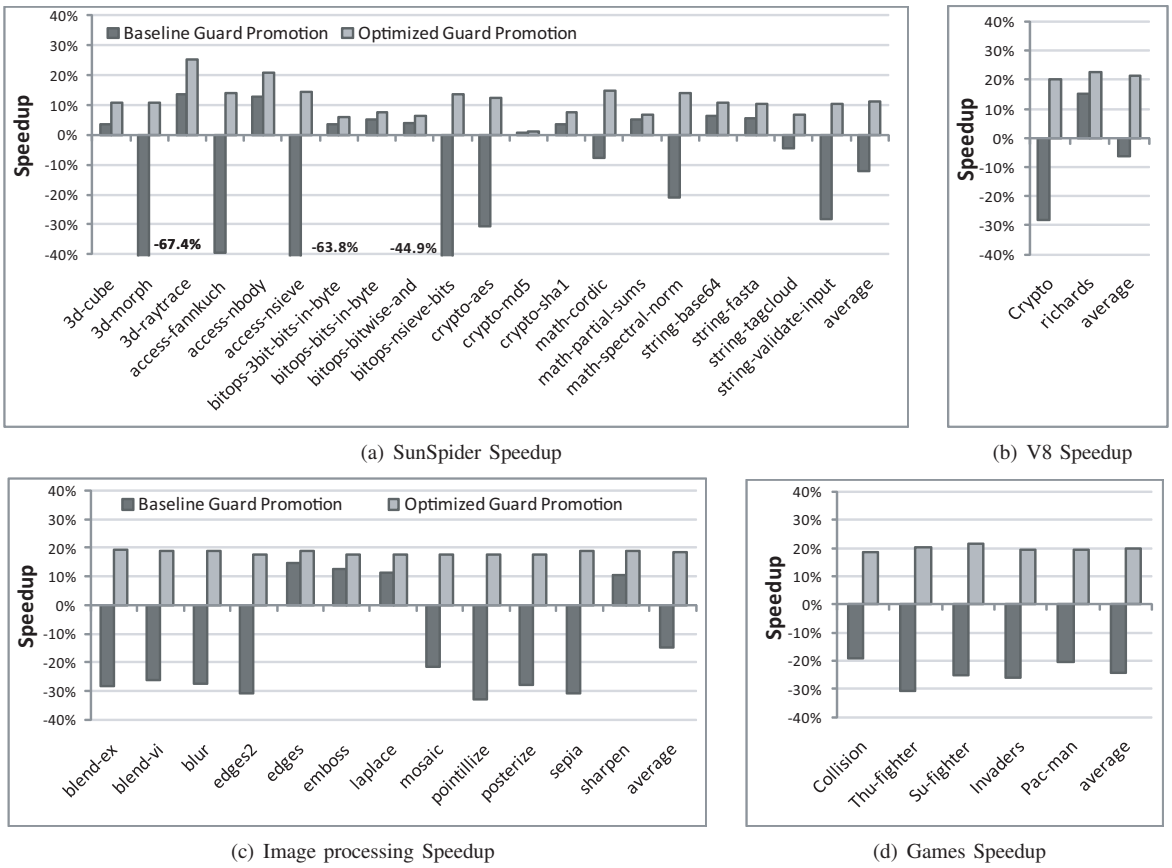
SunSpider has 26 JavaScript programs. However, TraceMonkey does not support recursion, the `eval` function, and regular expression `replace` operations, limiting the number of programs that can be properly traced [19]. Consequently, we excluded the following six benchmarks from our experiments: `controlflow-recursive`, `access-binary-trees`, `date-format-tofte`, `date-format-xparb`, `string-unpack-code`, and `regexp-dna`.

In the V8 suite, we excluded the `RegExp` benchmark due to its dependence on the regular expression library inside the engine rather than tracing. In addition, `DeltaBlue`, `RayTrace`, and `EarleyBoyer` perform poorly on the tracing JIT as only a small fraction of execution is spent running natively, mainly due to the lack of support for recursion in TraceMonkey. Therefore, we excluded them from our results as well. In this section, when we refer to SunSpider and V8, we mean these subsets of the suites. All experiments were performed on a system with an Intel Core i7 processor running at 3.20 GHz, and 4 GBs of main memory.

B. Results

Figure 9 presents the number of guards that passed the promotion heuristic and were moved to the ParaGuard trace. In addition to loop guards, which are always present in the main trace after guard promotion and are counted as non-promoted guards, most guards that check the integrity of various function return values (such as allocation functions) get rejected by the guard promotion heuristic. In order to move these guards, guard promotion either has to copy the corresponding function calls or move the return value directly using the buffers between the main and ParaGuard thread. Both of these approaches are inefficient, since they add overhead while only saving the guard comparison and branch on the main trace. However, many branch/case, overflow and mismatch guards successfully pass the heuristic and are moved to the ParaGuard trace. As can be seen, the ratio of moved guards varies between 25% and more than 80%.

Figure 10 shows the number of triggered guards in the ParaGuard trace, per 100,000 program instructions after applying guard promotion. This figure shows that many hot loops in these applications are type-stable and have infrequent changes in control-flow. This is the key to the effectiveness of the original tracing approach [19] and also the reason behind infrequent roll-backs from snapshots in our method. The majority of these triggered guards are branch guards after which ParaGuard rolls back the state and continues recording other paths of the branch in interpretation mode.



(a) SunSpider Speedup

(b) V8 Speedup

(c) Image processing Speedup

(d) Games Speedup

Fig. 11. ParaGuard speedup on 2 processors compared to the baseline tracing. The left bars show the speedup after guard promotion and the right bars show the speedup after applying the profile-based snapshot elimination optimization.

We originally applied guard branch aggregation to the ParaGuard trace. However, since the ParaGuard trace is shorter than the main trace in all benchmarks, in practice, applying this optimization proved ineffective on the overall performance. Furthermore, due to the infrequent number of side exits in these benchmarks (Figure 10), the drawback from identifying guard failures after N iterations rather than at each individual guard was negligible. Therefore, we present the performance results without applying guard branch aggregation.

Figure 11 shows the results of applying ParaGuard to the four benchmark suites on 2 processors, where one of them is running the main thread and the other one is running the ParaGuard thread. The left bars in this figure represent the speedup gained compared to TraceMonkey's sequential trace-based execution after applying guard promotion. The right bars show the resulting speedup after performing profile-based elimination of state snapshots.

Applying guard promotion by itself leads to an average slowdown of 12.2%, 0.1%, 14.7% and 24.2% on SunSpider, V8, image processing and gaming benchmarks, respectively, on two processors compared to the original tracing on one processor. The main reason for the slowdowns in these benchmarks is the large overhead of taking snapshots due to high number of individual array and heap accesses. In some of the benchmarks (16 out of 39 programs), where variable accesses are mostly scalar or multiple iterations update the same array or heap elements, the overhead of taking snapshots is much less and an average speedup of 8% is achieved.

After performing the profile-based snapshot elimination, all triggered guards during previous executions are kept in the main trace. The distribution of the number of these guards is similar to Figure 10. As can be seen in Figure 11, applying this optimization improves the performance of SunSpider, V8, image processing and game benchmarks to 11.2%, 21.4%, 18.3% and 19.8% over the baseline tracing, respectively. This improvement is mainly caused by the elimination of the snapshot taking process, and since the guard behaviors are quite stable with different inputs, the number of guards triggered in the ParaGuard trace after applying this optimization is close to zero. The main source of overhead in the execution is the synchronization between the main and the ParaGuard traces.

The highest variation in the profile-based promotion results exists in the SunSpider benchmark suite. This is mainly due to various ratios of promoted guards and also the non-uniform benefit from original tracing in these benchmarks. For instance, `crypto-md5` spends less than 20% of its total execution time in the native mode, and thereby, total performance benefit of our technique is around 1% in this benchmark. Overall, across the 39 benchmarks we studied, the ParaGuard technique achieves an average of 15% speedup over the original tracing technique.

Figure 12 shows the CPU utilization of the ParaGuard thread

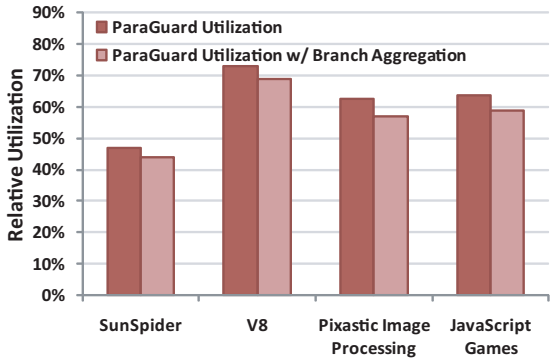


Fig. 12. Utilization of the ParaGuard thread relative to the main thread before and after applying guard branch aggregation optimization.

relative to the main thread with and without applying the guard branch aggregation optimization. The average utilization across all our benchmarks is 55% and guard branch optimization is able to reduce it to an average of 51%. This level of utilization shows a potential for using one processor for running ParaGuard threads in two JavaScript execution instances at the same time with each ParaGuard thread exploiting approximately half of the processing power in the extra core. Therefore, for instance, using 3 processors, two JavaScript programs can be accelerated with ParaGuard.

VI. Related Work

The idea of running traces for specializing hot code regions was proposed in the Dynamo binary rewriting system [17]. Dynamo utilizes run-time information to find hot patches and optimizes machine code accordingly. It also uses trace linking to connect traces together if possible. Our work is based on Mozilla's TraceMonkey, the trace-based JIT compiler described in [19] and released as a part of recent versions of Firefox [7]. TraceMonkey is able to achieve more than 10x speedup on some programs in the SunSpider suite compared to previous versions of SpiderMonkey on Firefox (which is an interpreter-only JavaScript engine). All this performance is achieved by type specialization and the tracing mechanism. Chang et al. [18] proposed a trace-based JIT compiler implemented on top of Adobe's Tamarin-Central (Tamarin-Tracing) which is their VM for implementing ActionScript and can execute JavaScript programs without any modifications. They also investigate using simpler opcodes in their IR and achieve up to 116% performance improvement over the non-traced code on SunSpider benchmarks. As we showed, by dynamically decomposing execution to main and ParaGuard traces and using extra resources in multicore systems, additional speedups can be achieved on top of tracing techniques on multicore systems. A recent proposal [20] presents a concurrent trace-based JIT in which the compilation from LIR to native code is performed as a background thread. This technique can achieve an average of 6% and a maximum of 25% speedup on the SunSpider benchmark suite. We choose a different approach and parallelize the execution by decoupling runtime checks rather than performing the compilation in parallel with the monitoring/recording. However, these two approaches are orthogonal and can be applied simultaneously.

SlipStream processors [25] speculate on certain code path and execute a pruned version of the program itself in parallel with the original execution. In SlipStream, the speculation support is provided by hardware. The Mitosis compiler [26] proposes a general framework to extract speculative threads as well as pre-computation slices (p-slices) that allow speculative threads to start earlier. MSSP [35] transforms code into master and slave threads to expose speculative parallelism. It creates a master thread that executes an approximate version of the program containing a frequently executed path, and slave threads that run to check results. All of these speculative multithreading works parallelize the main computation for purposes of prefetching or exploiting computational parallelism, where as in ParaGuard, we perform domain-specific runtime checks in parallel with the main computation in a dynamic language. Furthermore, in contrast to these works, we propose an all-software solution which works on commodity hardware. The LRPD test [29] performs runtime array tracking by using shadow arrays to follow exactly what array elements are touched in each thread. However, our accumulative array snapshot mechanism only keeps track of range of array accesses.

Several methods have been proposed for parallelizing runtime checks in static languages such as C/C++ [23], [24], [31], [33], [34]. Speck [24], FastTrack [23], ParExC [34], and Prospect [33] parallelize security checks, array bounds checks or data flow integrity checks by running the instrumented application in parallel with the original version in separate Linux processes. In these works, speculation is managed using heavy-weight, memory page-based speculation mechanisms at the OS kernel level. Due to the extremely high overhead of the runtime checks these works are looking into (e.g. upto more than 60x runtime overhead for dynamic memory checks in [23]), heavy-weight speculation and parallelization mechanisms could be used, at the cost of using a considerable number of extra processors. For instance, FastTrack [23] halves the overhead of the MudFlap memory safety instrumentation tool using 8 processors, though it is still several times slower than the original application. The technique proposed in [31] parallelizes information flow tracking using expensive extra hardware support which does not exist in commodity systems. However, in ParaGuard, we look into the guards inserted by a tracing compiler in a dynamic language, which poses a completely different set of challenges. Due to the relatively lower overhead of these checks and much tighter target performance constraints, we are not able to utilize heavy-weight speculation mechanisms as were used in those works. ParaGuard is the first software-only solution for offloading the extra checking overhead incurred by the runtime system to another thread in a dynamic language. A large portion of these checks (such as variable type checking) do not even exist in static language environments.

There is a significant amount of previous efforts in the area of memory speculation and transactional memory. Harris et al. goes through a detailed survey of different transactional

memory techniques in [21]. In particular, Shavit et al. proposed the first implementation of software transactional memory in [32]. The authors in [22], [16] proposed a lock-based approach where write locks are acquired when an address is written. Our rollback mechanism for taking interpreter snapshots in ParaGuard is a very low-cost and domain-specific checkpointing mechanism. Due to tight performance constraints, we were not able to exploit many ideas from the software memory speculation domain for ParaGuard's speculation and roll-back.

VII. Conclusion

As the web becomes the ubiquitous platform for execution of more complicated applications, a growing amount of computation is being handed-off to the client to minimize network traffic and improve user experience. The flexibility and ease of prototyping in the JavaScript language has made it the language of choice for most client-side web applications. However, as JavaScript applications are becoming larger and more computation intensive, there is more need for building high performance JavaScript engines in the client's browser. Trace-based JIT compilation is one approach towards tackling this issue. In this work, we proposed ParaGuard, which decouples execution from the runtime checks in a trace-based JavaScript engine and accelerates the execution by utilizing extra resources on multicore systems. We also introduced optimizations to further improve the performance. We showed that ParaGuard obtains an average of 15% speedup on two processors across 2 industry-standard benchmark suites, SpiderMonkey and V8, and two sets of JavaScript applications from the image processing and gaming domains.

Acknowledgements

We would like to thank Tim Harris for insightful discussions and feedback on this work. We extend our thanks to our shepherd, David Tarditi, and the anonymous reviewers for their valuable comments. We also thank Ganesh Dasika, Shuguang Feng, Shantanu Gupta and Amir Hormati for providing feedback on this work. This research was supported by the National Science Foundation grant CNS-0964478 and the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] "gameQuery - a javascript game engine with jQuery - http://gamequery.onaluf.org/."
[2] "Google V8 JavaScript Engine - http://code.google.com/p/v8."
[3] "http://gamequery.onaluf.org/demos/2/."
[4] "http://gamequery.onaluf.org/demos/3/."
[5] "http://gamequery.onaluf.org/demos/4/."
[6] "Invaders from Earth - http://www.senadbajramovic.com/game/."
[7] "Mozilla - Firefox web browser & Thunderbird email client - http://www.mozilla.com."
[8] "Online photo editing, online photo sharing - PhotoShop.com - http://www.photoshop.com."
[9] "PacMan 2 - http://www.masswerk.at/javapac/js-pacman2.html."
[10] "Picnik-Online photo editing in your browser - http://www.picnik.com."
[11] "Pixastic: JavaScript Image Processing - http://www.pixastic.com."
[12] "Spidermonkey engine - http://www.mozilla.org/js/spidermonkey/."
[13] "Sunspider javascript benchmark - http://www2.webkit.org/perf/sunspider-0.9/sunspider.html."
[14] "The Render Engine - http://www.renderengine.com/."
[15] "V8 Benchmark Suite - http://code.google.com/apis/v8/benchmarks.html."
[16] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory," in *Proc. of the '06 Conference on Programming Language Design and Implementation*, 2006, pp. 26–37.
[17] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proc. of the '00 Conference on Programming Language Design and Implementation*, 2000, pp. 1–12.
[18] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz, "Tracing for web 3.0: trace compilation for the next generation web applications," in *Proc. of the 2009 international conference on Virtual Execution Environments*, 2009, pp. 71–80.
[19] A. Gal *et al.*, "Trace-based just-in-time type specialization for dynamic languages," in *Proc. of the '09 Conference on Programming Language Design and Implementation*, 2009, pp. 465–478.
[20] J. Ha, M. Haghighat, S. Cong, and K. McKinley, "A concurrent trace-based just-in-time compiler for JavaScript," University of Texas, Austin, Tech. Rep. TR-09-06, Feb. 2009.
[21] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Edition*. Morgan & Claypool Publishers, 2010.
[22] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions," *Proc. of the '06 Conference on Programming Language Design and Implementation*, vol. 41, no. 6, pp. 14–25, 2006.
[23] K. Kelsey, T. Bai, C. Ding, and C. Zhang, "Fast track: A software system for speculative program optimization," in *Proc. of the 2009 International Symposium on Code Generation and Optimization*, 2009, pp. 157–168.
[24] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008, pp. 308–318.
[25] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," in *Proc. of the 33rd Annual International Symposium on Microarchitecture*, 2000, pp. 269–280.
[26] C. G. Quinones *et al.*, "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," in *Proc. of the '05 Conference on Programming Language Design and Implementation*, Jun. 2005, pp. 269–279.
[27] A. Raman, H. Kim, T. R. Mason, T. Jablin, and D. August, "Speculative parallelization using software multi-threaded transactions," in *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 65–76.
[28] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "Jsmeter: Characterizing real-world behavior of javascript programs," Microsoft Research, Tech. Rep. MSR-TR-2009-173, Dec. 2009.
[29] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 2, pp. 160–180, 1999.
[30] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *Proc. of the '10 Conference on Programming Language Design and Implementation*, 2010, pp. 1–12.
[31] O. Ruwase, P. Gibbons, T. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan, "Parallelizing dynamic information flow tracking," in *SPAA '08: 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2008, pp. 35–45.
[32] N. Shavit and D. Touitou, "Software transactional memory," *Journal of Parallel and Distributed Computing*, vol. 10, no. 2, pp. 99–116, Feb. 1997.
[33] M. SuBkraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer, "Prospect: a compiler framework for speculative parallelization," in *Proc. of the 2010 International Symposium on Code Generation and Optimization*, 2010, pp. 131–140.
[34] M. SuBkraut, S. Weigert, U. Schiffel, T. Knauth, M. Nowack, D. Brum, and C. Fetzer, "Speculation for parallelizing runtime checks," in *Proc. of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2009, pp. 698–710.
[35] C. Zilles and G. Sohi, "Master/slave speculative parallelization," in *Proc. of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002, pp. 85–96.