Prospect: A Compiler Framework for Speculative Parallelization

Martin Süßkraut
Technische Universität Dresden
suesskraut@se.inf.tu-dresden.de

Thomas Knauth
Technische Universität Dresden
thomas@se.inf.tu-dresden.de

Ute Schiffel
Technische Universität Dresden
ute@se.inf.tu-dresden.de

Stefan Weigert
Technische Universität Dresden
stefan@se.inf.tu-dresden.de

Martin Meinhold
Kontext E GmbH
m.meinhold@kontext-e.de

Christof Fetzer
Technische Universität Dresden
christof@se.inf.tu-dresden.de

Abstract
Making efficient use of modern multi-core and future many-core CPUs is a major challenge. We describe a new compiler-based platform, Prospect, that supports the parallelization of sequential applications. The underlying approach is a generalization of an existing approach to parallelize runtime checks. The basic idea is to generate two variants of the application: (1) a fast variant having bare bone functionality, and (2) a slow variant with extra functionality. The fast variant is executed sequentially. Its execution is divided into epochs. Each epoch is re-executed by an executor using the slow variant. The approach scales by running the executors on multiple cores in parallel to each other and to the fast variant. We have implemented the Prospect framework to evaluate this approach. Prospect allows custom plug-ins for generating the fast and slow variants. With the help of our novel StackLifter, a process can switch between the fast variant and the slow variant during runtime at arbitrary positions.

Categories and Subject Descriptors D.3.4 [Processors]: Compilers; D.3.4 [Processors]: Optimization

General Terms Design, Experimentation, Measurement, Performance, Security

Keywords parallelization, speculation, bounds checker, assertions, stack translation

1. Introduction
Despite many years of research, making efficient use of modern multi-core and future many-core CPUs is a major challenge. One promising approach comes from the hardware community [Zilles and Sohi 2002] which we call the predictor/executor approach. Two variants of an application are generated: a fast and a slow variant. The slow variant uses more CPU cycles than the fast one because the slow variant is, for example, made more robust with the help of additional error checks and error handling. The goal is to provide the functionality of the slow variant and the runtime of the fast variant. This is achieved by running the fast variant on one core and cutting its execution into epochs. Each epoch is re-executed using the slow variant on another core. This approach scales well with the number of cores as long as the re-execution of the epochs can be decoupled from each other. The maximum speedup is determined by the ratio of the execution times of the slow and the fast version. This approach has gained attention in the context of runtime checking [Nightingale et al. 2008; Ruwase et al. 2008; Süßkraut et al. 2009; Wallace and Hazelwood 2007]. The fast variant is the original application and the slow variant performs runtime checks that are added, as far as we know, via dynamic binary instrumentation (DBI). The parallelization reduces the user perceived overheads of the runtime checks. The use of DBI introduces, however, high runtime overheads even for simple checks. This means that one needs additional cores to hide the overheads through a higher degree of parallelization. However, gaining good speedups is not easy. For example, the parallelized taint analysis of [Nightingale et al. 2008] has a similar runtime overhead as taint analysis without parallelization [Newsome and Song 2005].

FastTrack [Kelsey et al. 2009] introduces fast-track regions. For each such region, a fast and a slow variant is given, either manually by the programmer or generated by instrumentation with a compiler plug-in. Due to the use of compile time instrumentation, FastTrack avoids the DBI-related performance issues. Most importantly, FastTrack enables additionally the instrumentation of the fast variant. But FastTrack only supports the approach on defined regions. This introduces composability problems: How should one deal with a fast-track region nested in the fast or slow path of another fast-track region? Furthermore, FastTrack does not support speculation for system calls like, for example, [Nightingale et al. 2008].

In Prospect, we combine the advantages of FastTrack with those of the predictor/executor approach. Prospect allows the integration of compiler plug-ins for generating both a fast and a slow variant from the original application. The compiler plug-ins can instrument the whole application, i.e., there are no special regions. In particular, we support system calls within the variants - similar to the approach by [Nightingale et al. 2008]. Regarding the system call speculation, the main difference to [Nightingale et al. 2008] is that the slow and the fast variant can perform different system calls. For example, to speed up the fast variant, one might execute only a subset of the system calls executed by the slow variant. In this paper, we will mainly focus on Prospect’s compiler framework and not on the speculative execution of system calls.
Our main contribution is the Prospect StackLifter. The StackLifter is a compile time instrumentation to allow switching from the fast variant to the slow variant at given points in the application. It is required that these points are known at compile time. The StackLifter solves two major problems that need to be addressed when switching between code bases: the variants differ in (1) the machine register allocation, and (2) the stack layout. Due to the different instrumentations of the fast and slow variant, the machine code of both variants will be different. For a given common point in both variants, machine registers may be used to hold different values / variables. For example, values held in registers in the fast variant might be temporarily stored on the stack in the slow variant. This might happen because of a higher register pressure. The slow variant executes additional code and accesses additional variables.

In Section 3, we present the StackLifter. We also describe two compiler plug-ins: (1) an out-of-bounds checker similar to [Kelsey et al. 2009] in Section 4.1, and (2) an optimizer that generates a fast variant for user-defined sanity checks in Section 4.2. We evaluate Prospect in Section 5. We describe the related work in Section 6 and Section 7 concludes the paper.

2. Prospect Overview

Prospect uses the predictor/executors approach of [Kelsey et al. 2009; Nightingale et al. 2008; Süßkraut et al. 2009; Zilles and Sohi 2002] to parallelize an application. Figure 1 (i) shows a fast variant and a slow variant derived from the same code base. Our goal is to provide the functionality of the slow variant while not exceeding the fast variant’s runtime. To achieve this, the Prospect framework parallelizes the execution of the slow variant (see Figure 1 (iii)). At runtime, it executes the fast variant in a predictor process – which is used to compute future states of the slow variant. The execution of the predictor is partitioned into epochs. The state of the predictor at the start of an epoch is used to spawn an executor process. The executor re-executes an epoch using the slow code variant and the predictor state (which was obtained using the fast code variant). We can parallelize the application by running the individual executors and the predictor in parallel. The maximum possible speedup is the execution time of the slow variant divided by the execution time of the fast variant.

At each epoch boundary, Prospect takes a snapshot of the fast variant. The snapshot is similar to a UNIX fork. The fast variant continues its execution. The forked executor switches from the fast variant to the slow variant and starts executing the epoch in the slow variant. At the end of an epoch, the slow variant terminates, whereas, the fast variant forks the next epoch.

Prospect is not completely transparent for the application developer. The application developer has to ensure that the application calls prospect_chkpt periodically and preferably, with a constant frequency. This function starts a new epoch and is provided by the Prospect runtime.

Prospect has two major components:

- The **Prospect compiler** which generates the fast and/or slow variant of the given application, and
- The **Prospect runtime** which provides deterministic replay for re-executing the slow variant in the executors and speculative execution for the fast variant in the predictor.

2.1 Compiler Infrastructure

The slow and fast variant are generated by the Prospect compiler from the original application. We consider three cases of variant generation:

- The original application is the fast variant. The slow variant is generated by adding additional code (like runtime security checks) to the original application [Nightingale et al. 2008].
- The original application is the slow variant. The fast variant is generated by removing code from the original application. For instance, aggressive but potential unsafe optimizations can remove code [Kelsey et al. 2009].
- The first and the second approach can by combined, i.e., both variants are generated from the original application.

Figure 2 shows the work-flow of the Prospect compiler. Our novel StackLifter (Section 3) takes the original application’s code and generates the initial versions of the fast and the slow variant.

---

**Figure 1.** Our parallelization approach executes a fast variant on one core. The execution of the fast variant is partitioned into epochs. Each epoch is re-executed with a slow variant with more functionality. The re-execution happens in parallel to the fast variant on multiple cores.

**Figure 2.** The Prospect work-flow: The StackLifter generates the two initial code bases for the fast and the slow variant. Both variants can be instrumented to remove or add functionality, respectively. Both variants are linked together with a generated framework code that manages the switch from the fast variant to slow variant at epoch boundaries.
In particular, it prepares both variants in a way that it is possible to switch from the fast variant to the slow variant at epoch boundaries.

Both variants can then be instrumented, e.g., to remove existing functionality that does not affect the state or add error handling functionality (Section 4). The instrumentation must preserve the state equivalence property:

**Definition 1.** The application state of the fast variant at the end of epoch \(e\) must be equivalent to the application state of the slow variant at the end of \(e\).

This property ensures that from an external point of view, the parallel execution of the slow variant is equivalent to the sequential execution of the fast variant. Note, that we require neither the heap nor the stack of the fast and the slow variant to have the same values at the end of epoch \(e\). However, there has to exist a bidirectional mapping from the state of the fast variant to the state of the slow variant. The mapping does not need to be defined. It only needs to exist. Currently, we do not enforce the state equivalence property. However, one could implement the bidirectional mapping and use the mapping to check at the end of each epoch if the state equivalence property still holds. We show in Section 4.1.2 how to partly circumvent the state equivalence property by speculative variables.

In our experience, the instrumentation process does not need be aware of the parallelization.

Prospect also generates framework code that connects the code bases of both variants. The framework code contains primarily a new `main` function that sets up the Prospect runtime, any additional runtime that was added for the slow variant and it starts the first epoch.

We have implemented Prospect using the LLVM compiler framework [Lattner and Adve 2004]. StackLifter and the instrumentation for the fast and slow variant are LLVM compiler passes.

### 2.2 Runtime Support

Prospect’s runtime support consists of speculative execution and deterministic replay of system calls and speculative variables to manage additional state in the slow variant.

#### 2.2.1 Speculative Execution and Deterministic Replay of System Calls

Prospect performs external actions of the fast variant speculatively [Nightingale et al. 2008]. For instance, `write` system calls are held back until they are re-executed by the slow variant. Hence, we ensure that external actions only become visible after the slow variant has verified them.

To support the state equivalence property, we use deterministic replay [Srinivasan et al. 2004] for re-executing the slow variant. Prospect records all non-deterministic external events that happen in epoch \(e\) for the fast variant. When the slow variant is re-executed for epoch \(e\), Prospect replays the recorded events. For example, the time value returned by the `gettimeofday` system call in the fast variant is also returned in the slow variant for the same call to `gettimeofday`.

We have implemented the speculative execution and deterministic replay for Linux with a kernel module, similar to Speck [Nightingale et al. 2008]. Speck provides system wide speculation, i.e., external actions of the fast variant are speculative propagated to other processes. In case of an abort, all processes containing speculative state are rolled back. In contrast to Speck, our kernel module isolates multiple applications running under Prospect at the same time from each other.

Currently, to support deterministic replay we have to limit Prospect to single-threaded applications. Recording and deterministically replaying the non-deterministic scheduling decisions of the OS with a low performance overhead is still an unsolved problem.

However, since Prospect parallels the slow variant of applications, we believe that this limitation is unimportant. Any progress made in the field of deterministic replay can be incorporated into Prospect in the future. More information on speculation and deterministic replay can be found in related work [Nightingale et al. 2008; Srinivasan et al. 2004].

#### 2.2.2 Speculative Variables

Some slow variants need to track state at runtime that does not exist in the fast variant. For instance, if the slow variant adds out-of-bounds checks, it needs to keep track of the bounds of all allocated buffers at runtime. This additional state violates the state equivalence property. There are two solution to this problem:

- One could track the state in both variants, but use it in the slow variant only. The advantage of this approach is that the state of the fast variant is exactly the same as the state of the slow variant. But, the tracking of the state slows down the fast variant. Figure 1 shows that the parallelization speedup of our approach depends on the relation between the runtimes of the fast and the slow variant. The faster the fast variant and the slower the slow variant the larger the parallelization speedup. Thus, we do not want to track any additional state in the fast variant.
- The slow variant can track the additional state in speculative variables previously introduced in [Süßkraut et al. 2009]. Speculative variables completely eliminate the need to track additional state in the fast variant too. We show in Section 4.1.2 how we track the state of out-of-bounds checks with speculative variables.

### 3. StackLifter

One main goal of Prospect is to enable compiler plug-ins to instrument slow and fast variants independently of each other. The issue is that we need to switch from the fast variant to the slow variant at the start of an epoch. This is difficult because, for instance, a plug-in might remove some temporary variables from the fast variant and add new temporary variables to the slow variant. The StackLifter’s purpose is to instrument the application code to enable a switch from the fast to the slow variant.

A fast function \(F_{\text{fast}}\) and its slow equivalent \(F_{\text{slow}}\) may have different stack layouts. Figure 3 illustrates this for functions `main`, `F`, and `G`. After the StackLifter run, a compiler plug-in changed the order of variable definitions in `main_slow` and added a new variable `A to G_{slow}`.

After a new epoch has been spawned and before continuing execution of the slow variant, the application’s call stack must be translated. To be transparent to the application, the call stack has to look...
like as if only slow functions had been executed. After translating
the call stack, execution can continue normally. Global data and the
heap do not need to be translated, as the state equivalence property
requires the compiler plug-ins not to change the heap layout or the
layout of global data.

Stack translation begins with setting the doUnwind flag. The
call stack is traversed up to the outermost function, i.e., usually
to main, saving all necessary information for each stack frame to
allow reconstruction (Figure 3 step (1)). Once the top of the stack
is reached (step (2)), we rebuild the call stack by calling the slow
variant of each function. We use the saved information to rebuild
the call stack (step (3)). When reaching the point where the stack
lifting was triggered, execution resumes normally.

We use LLVM to perform the necessary modifications. All code
modifications are performed statically. The input and output
format is LLVM intermediate representation (IR). Therefore, our
StackLifter is independent of the underlying hardware platform.

3.1 Example
For an application programmer, the main difference between Fast-
Track and Prospect is the programming interface. In FastTrack, the
programmer specifically starts and ends a fast-track region. This is
translated to an if branch, where one branches is executed in the
fast variant and the other in the slow variant.

In Prospect, we support application wide instrumentation and
do not want to restrict instrumentation to a set of regions. In our
current implementation, the programmer has to mark possible places
where a new epoch could be spawned by calling prospect_chkptnt. Listing 1 shows an example. Like fork, a call to
prospect_chkptnt from the fast variant returns twice: (1) in the fast variant, and (2) in the slow variant. When pros-
pect_chkptnt is called from the slow variant, the re-execution
of the current epoch is finished. At runtime, when prospect_chkptnt
is called from the fast variant and returns into the slow variant, it
needs to switch from the fast variant’s code base to the slow vari-
ant’s code base.

```
Listing 2.
Function call transformation for fast variant
call1.i: : name to id block as a function call site
call void @foo()
    : check if doing an unwind
    %doUnwind = load i8* @doUnwind
    %doUnwindCmp = icmp eq i8 %doUnwind, 0
    : branch to register saving code or continue
    : executionnormally
    br i1 %doUnwindCmp, label %call1_succ.
    label %save_regs,1
```

The StackLifter ensures that all slow functions only call slow
functions. Once switched to slow execution, we do not want to
leave the set of slow functions. Direct function calls are easily
changed by simply altering the name of the called function. The
target of an indirect function call, though, can only be changed
at runtime. Hence, StackLifter inserts additional code before each
indirect function call (see Listing 3):

```
Listing 3.
Indirect call transformation for slow variant
call1.i:.
%castedPtrToOrig = bitcast void (*)%orig to i8*
    ; @fp2sp translates fast function pointer
    ; to slow variant’s function pointer
%ptrToClone = call i8* @fp2sp (i8 %castedPtrToOrig)
%castedPtrToClone = bitcast i8* %ptrToClone to void (*)
call void %castedPtrToClone () nonnull
```

We call a function fp2sp which, given a slow/fast function’s
address, will return the address of the corresponding slow function.
If the passed pointer already points to a slow function, fp2sp
simply returns it. fp2sp will use the input function pointer as
an index into a map. This map is constructed before executing the
main function by code generated by the StackLifter.

3.2 Implementation Details
StackLifter clones all functions defined in a given module. Each
function now comes in two flavors: (i) a fast, and (ii) a slow variant.
The slow variant’s name will be the original name appended with an
unique suffix, e.g., originalName_slow. Hence, all functions
ending in _slow belong to the set of slow functions. The predictor
executes the fast variants and the Executors the slow variants. An
executor receives its initial state from the predictor. Hence, all
stack frames on the stack belong to fast versions. Prospect needs
to replace the stack frames of the fast versions by their _slow
counters.

StackLifter creates a new basic block for each function call. The
purpose is to allow us to use the basic block as a branch label.
Control flow can be diverted to any function call. This is neces-
sary when reconstructing the call stack after an unwind. The basic
block only contains an LLVM call instruction and an unconditional
branch instruction. The branch will jump to the next instruction
after the call as defined by the original application. Basic blocks
will be given a unique name to identify them as “function call” ba-
sic blocks, e.g., callX where X is a running number. The basic
block reached via the unconditional branch instructions will carry
the same name as the function call basic block plus a suffix, e.g.,
callXsucc.

The basic block of a function call in the fast variant carries
two additional instructions. Upon return from each function call,
the global doUnwind flag is checked. If the check fails, control
flow continues as in the original program. If the check succeeds,
execution continues with a register saving basic block (discussed in
Section 3.3). Listing 2 has LLVM code for a transformed function
call of the fast variant of a function.

```
Listing 1. API example.
int foo() {   prospect_chkptnt(); ... }
void bar(char *b) {
    assert(b);
    int i = foo();
    return b[i];
}
```

For the example in Listing 1 a jump from the fast code base into
the slow code base (right behind the call to prospect_chkptnt) is
not enough. Consider, an instrumentation of the fast variant, that
removes the assert in line 3. The slow variant b might be loaded
into a machine register in line 3. When foo returns b is expected to
be in this register in line 5. Whereas, in the fast variant line 3 does
not exist and when foo returns b is still on the stack. That means
after prospect_chkptnt returns into the slow variant b has to
appear in the right register. Our approach is that both variants are
prepared by the StackLifter before the compiler plug-ins instrument
them.

3.3 Saving Registers
During a stack translation, each stack frame of a fast function must
be replaced by an equivalent stack frame of its slow variant. Hence,
register saving code is inserted into the fast functions. During
a stack translation – when unwinding the call stack of the fast
functions – for each function F, the state of F_{fast} is stored in a
buffer. This buffer permits the reconstruction of the state in F’s
slow counterpart F_{slow}. The storing of the state is done in the
register saving basic block.

For each function call within a fast function, there is a separate
register saving basic block. In LLVM, the state of function F at
instruction 1 is represented by the values of all live registers in F at 1. After each call, different registers might be live. Hence, each function call has its own register saving basic block. We perform liveness analysis for every basic block containing at least one function call. All registers marked live on entering the function call basic block need to be preserved.

An example of a register saving basic block is shown in Listing 4. External helper functions (push164 and pushFloat) are called to store all live registers (%reg1 and %reg2) in a separate buffer. Additionally, a label, uniquely identifying the function call basic block, is stored too (pushLabel). This label is of importance again, when the stack frame is reconstructed. The end of each register saving basic block is marked by a simple return instruction. Stack unwinding then continues in the caller.

Listing 4. Save register block
```
save_regs_3:
call void @push164(i64 %reg1)
call void @pushFloat(float %reg2)
call void @pushLabel(i32 1)
ret i32 undef
```

The buffer holding live registers and labels is organized as a stack. The register saving block pushes all live registers to this stack. In the slow function all live registers are restored from the stack in reverse order.

### 3.4 Restoring Registers

While a fast function needs to save the live registers, its slow function needs to be able to restore the live register with the help of a register restoring basic block. As with register saving basic blocks, there is a restore basic block for each function call. When entering a slow function, we need to check if this is a call because of a stack reconstruction. This is done by calling the external helper function popLabel. It returns zero, if no reconstruction is going on. Otherwise, the return value is a label identifying a specific function call basic block in the current slow function. A switch statement will divert control to the correct restore basic block based upon the return value of popLabel. Listing 5 shows an example of a slow function’s entry basic block.

Listing 5. New entry block
```
new_entry:
  %next_label = call i32 @popLabel()
  switch i32 %next_label, label %old_entry [ i32 1, label %restore_regs_1
  i32 2, label %restore_regs_2
  ]
```

If no reconstruction goes on (%next_label is zero) execution continues at original entry basic block %old_entry.

Listing 6 shows an example of a register restoring basic block.

Listing 6. Restore register block
```
restore_regs_3:
  %reg2 = call i64 @popFloat()
  %reg1 = call i64 @pop164()
  br label %call2inv_3_slow
```

The basic block calls an external helper functions (pop164 and popFloat) to retrieve the value of a live register (compare with Listing 4). After all live registers have been restored, execution continues by branching to the function call basic block identified by the label pop at function entry. Restoring then continues with the function called in the function call basic block.

3.5 Restoring Static Single Assignment Form

LLVM uses Static Single Assignment (SSA) in its LLVM intermediate representation (IR). All modifications need to preserve SSA. Our transformations temporarily violate the SSA constraint. This section describes how the SSA form is restored.

Conceptually, we use a modified version of the algorithm described by [Cytron et al. 1991]. One assumption made in the algorithm for placing Φ nodes is that all variables are defined and initialized in the function’s entry basic block [Cytron et al., pg. 25]. This, however, does not hold for programs in LLVM IR. The LLVM IR is compiled from C code, where variables can be initialized nearly anywhere in the function.

Using the unmodified algorithm proposed in [Cytron et al., pg. 25] can lead to wrongly placed Φ nodes. See Figure 4 for an example. Registers v_1 and v_2 refer to the same variable v. Variable v is defined and initialized in basic block X, and only used in basic block W. Following the algorithm in [Cytron et al., 1991], a Φ-node for v should be inserted in Y. First, v’s scope is limited to basic blocks X and W. Hence, the Φ-node in Y has no uses. Second, because variable v was defined and initialized in X there is no incoming value for this Φ-node for the incoming basic block Z. If v would have been defined and initialized in basic block entry, as assumed by [Cytron et al., 1991], there would be an incoming value. However, as v’s scope is limited to basic blocks X and W, the Φ-node in Y is neither possible nor needed.

Our solution is:

1. Apply the algorithm from [Cytron et al. 1991].
2. Remove any illegal Φ-nodes. We use a fix point algorithm and iteratively remove Φ-nodes. If, during one iteration, we find no Φ-nodes to remove, the algorithm terminates. We use three rules to detect an illegal Φ-node. A Φ-node is deleted if:
   (a) It is never used.
   (b) Its uses are exclusively incoming values for itself.
   (c) The definition of at least one of its incoming values does not dominate the basic block for which the incoming value is specified.

Rule a) is self-explanatory [Briggs et al. 1998]. Rule b) is a special case of Rule a). A Φ-node according to Rule b) has uses (but only itself). Therefore, it cannot be removed according to Rule a). But, as the Φ-node is not used by any other instruction it is nevertheless not needed.

Consider the control flow graph in Figure 4 as an example for Rule c). As a default incoming value for each direct predecessor of Y, the original definition of v_1 is inserted, i.e.:

```
%v_2 = phi i32 [v_1, label %W], [v_1, label %Z])
```

When updating incoming values later, the pair [%v_1, label %Z] would remain unchanged. As explained above, this Φ-node is illegal. Because of Rule c) we remove this Φ-node. The reason is that the basic block X (where the incoming value %v_1 is defined), does not dominate basic block Z (the basic block for which the incoming value is defined).
3.6 Stack-local Variables
Addresses of stack-local variables can change during stack translation because additional local variables might be present in the slow variant’s function. Therefore, we put all addressable variables on a separate alloca stack. This stack is not changed by stack translation. Hence, the address of any variable on the alloca stack is the same for the fast and for the slow variant.

Technically, we replace all LLVM alloca instructions with our own implementation that allocates the variables on the alloca stack. On function entry, we store the current frame address of the alloca stack. And on each function exit, we restore the previously stored frame address.

3.7 Integration with prospectchkpt
At runtime, an application spawns a new epoch by calling prospectchkpt_fast in the fast variant. After forking the new epoch, the flag doUnwind is set in the executor and prospectchkpt_fast returns to its caller func_fast. Because doUnwind is set, the register saving block of func_fast is executed (see Sections 3.2 and 3.3). After the registers of func_fast are saved, func_fast returns to its caller. Again, the registers of func_fast’s caller are saved and the function returns. The process continues iteratively saving the registers of all function frames on the stack, including main_fast.

Function main_fast returns to our generated main. Our main now calls main_slow with the doUnwind flag still set. In main_slow the register restoring block is triggered (see Section 3.4). This restoration process continues iteratively until func_slow(func_fast’s counterpart in the slow variants code base) is reached. Function func_slow restores its registers and calls prospectchkpt_slow. In prospectchkpt_slow the flag doUnwindis cleared and it returns back into func_slow. Then the execution continues normally in func_slow in the slow variant.

4. Prospect Plug-ins
To evaluate the Prospect framework, we implemented two plug-ins: (1) Out-of-Bounds (OOB) instruments a slow variant with additional out-of-bounds checks for each memory access, and (2) FastAssert removes all asserts from a fast variant. We use the two plug-ins to evaluate the performance of our framework. We did not try to push the state-of-the-art of out-of-bounds checkers.

Because the plug-ins are applied after the StackLifter (see Figure 2), the StackLifter’s instrumentation is visible to the plug-ins. In general, we found that these instrumentations are transparent to the plug-ins as the instrumented code is valid LLVM. It is even possible to change the restored state. In order to do so, a plug-in can wrap the restoring helper functions, e.g., popI64 and popFloat.

4.1 Out-of-Bounds checks
In our experience, Prospect permits runtime checks to be added by a plug-in in almost the same way as one would add it to a sequential program. To justify this claim, we first show briefly how our simple OOB checker is implemented without Prospect and then how we adapted the OOB checker for Prospect to parallelize its runtime checks. Our goal is in both cases to detect out-of-bounds accesses to heap allocated buffers.

4.1.1 OOB without Prospect
The OOB plug-in is implemented as an LLVM pass. At compile time, the OOB plug-in adds a runtime check for every memory access. To keep track of all currently allocated buffers, our instrumentation wraps all malloc and free calls of the application.

The OOB checks fail if and only if, the checked memory access goes to the heap but not into a currently allocated buffer. In LLVM, for most memory accesses the reference of the corresponding base addresses (start of an allocated buffer) can be identified at compile time. Our current prototype only instruments such memory accesses with checks. To do so, we keep the size and base-addresses of allocated buffers in a hash map at runtime. For every malloc call, a new entry is added to the map. Consequently, for every free call, the corresponding entry is removed from the map.

4.1.2 OOB with Prospect
The instrumentation for OOB with Prospect is the same as the instrumentation without Prospect except that only slow functions are instrumented. The fast variant does neither update the map nor does it perform any checks. The major problem is that the slow variant does not know which blocks are allocated. Prospect supports speculation to address this problem. For example, in Listing 7, the allocation and the memory access happen in different epochs e0 and e1+1, respectively. Because, the slow variants of e0 and e1+1 are executed in parallel to each other, the slow variant of e1+1 might access variable buf and check its size, before e0 allocates buf and stores the size of buf in the hash map.

Listing 7. Allocation and memory access in different epochs.
1 char *buf = malloc(20);
2 prospectchkpt();
3 buf[0] = ‘h’;

Speculative Variables We solve this problem by using speculative variables [Süßkraut et al. 2009]. At runtime, each epoch starts with an empty hash map to store the start address and the size of each allocated buffer. When an OOB check fails, the runtime check adds a speculative entry into the hash map. We call this speculative entry a speculative variable. The speculative variable contains the expected buffer bounds derived from the checked memory access. For example, in Listing 7 in line 3 an entry (&buf, 1) is inserted into the hash-map of epoch e1+1. Subsequent accesses may update the speculative variable if the expected buffer bounds have to be extended. Allocations with malloc create non-speculative entries. If the allocation happens before the memory access in the same epoch, no speculative variable is created. Note that the first epoch will not create speculative variables: for every memory access it must have already seen the allocation.

At the end of each epoch ei+1, all speculative variables are verified against the hash map hi of the previous epoch ei. For each speculative variable sp in hi+1 there must be an entry in hi for which the expected bounds in sp can be verified. If no such entry is found, the application is aborted. After all speculative entries are verified they become non-speculative entries. The non-speculative entries of hi are merged into hi+1. By induction, epoch ei+2 can then be verified against hi+1 of epoch ei+1. We give more details about speculative variables in [Süßkraut et al. 2009].

4.2 FastAssert
Software developers are encouraged to add runtime assertions to their source code [Meyer 2000]. One of the trade-offs of runtime assertions is their runtime overhead. FastAssert (partly) mitigates the negative effects of assertions on the application runtime. The plug-in removes any assertions and functions that neither change the internal nor the external application state from the fast variant. The slow variant still contains the assertions. Hence, assertions...
<table>
<thead>
<tr>
<th>Benchmark/Application</th>
<th># calls to prospect_chkptnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacation</td>
<td>3</td>
</tr>
<tr>
<td>Genome</td>
<td>4</td>
</tr>
<tr>
<td>Whetstone</td>
<td>6</td>
</tr>
<tr>
<td>LinPack</td>
<td>2</td>
</tr>
<tr>
<td>bzip2</td>
<td>8</td>
</tr>
<tr>
<td>BOOST’s words</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1. The number of calls to prospect_chkptnt that we inserted into the source code per used benchmark/application.

are still checked at runtime, but their computational overhead is parallelized.

For each function $f$, FastAssert computes if $f$ might change the internal or external state of the application or if $f$ transitively calls a function that might change the internal or external state of the application. Internal state changes are identified by store instructions. External state changes are identified by calls to external functions. If a function $f$ does neither, calls to $f$ are removed from the fast variant. The external function assert_fail is a special case. It is used to implement the assert macro on our platform. Therefore, assert_fail is considered as not changing any state.

By removing not only assertions itself but also side-effect free computation, we also remove user defined sanity checking code. We expect that FastAssert not only mitigates the perceived performance overhead of existing assertions, but also motivates to include more (computationally expensive) assertions.

5. Evaluation

Our evaluation focuses on the speedup achieved by Prospect and the overheads introduced by Prospect's components. We performed all measurements on a DELL PowerEdge 1950 with 2x Intel XEON E5430 (8 cores) and 16 GB RAM. Each data point is the average of at least three measurements.

5.1 Parallelization Speedup

We measured the parallelization speedup on four benchmarks and two real world applications for our compiler plug-ins out-of-bounds and FastAssert. In order to make the benchmarks and applications parallelizable with Prospect we had to split them into epochs. Effectively, we had to add calls to prospect_chkptnt to the four benchmarks and two applications. We have two rules of thumb to place calls to prospect_chkptnt:

- If the benchmark or application consists of multiple stages, we place the call to prospect_chkptnt between stages with big workloads.
- We also added calls to prospect_chkptnt into loops with big workloads.

Both rules can be combined, e.g., if a stage contains a loop with big workloads.

Table 1 shows the number of calls to prospect_chkptnt that we inserted into the source code per used benchmark/application. We used the first five benchmarks/applications to evaluate our out-of-bounds instrumentation and BOOST’s words to evaluate FastAssert.

5.1.1 Out-Of-Bounds

To measure the speedup of parallelizing the bounds checks with Prospect, we used five different benchmarks from different application domains. First, Genome and Vacation are part of the STAMP [Cao Minh et al. 2008] benchmark suite for transactional memory. The performance of both benchmarks is CPU and memory-bound. All STAMP benchmarks can be executed in parallel using multiple threads. In [Süßkraut et al. 2009] we compare the scalability of the STAMP benchmarks using Prospect with parallelizing the slow variant of the STAMP benchmarks using Software Transactional Memory. We found that the parallelization with Prospect scales better. We ascribe this effect to increased contention which limits scalability. In slow variant the contention between concurrent transactions is higher than in the fast variant. For more details we refer to [Süßkraut et al. 2009].

However, for this experiments we run the STAMP benchmarks single-threaded since we want to show the parallelization with Prospect. Second, we used Whetstone and LinPack. Both come from the high performance community and are, at least in our measurements, CPU-bound only. The last application is bzip2, a real application and not a benchmark.

Figure 5 shows the runtime (in s) of the five benchmarks. We run all benchmarks in four configurations:

1. without out-of-bounds checks (OOB) and without Prospect to show the lower bound for the runtime,
2. without OOB but with Prospect to show the framework’s overhead,
3. with OOB and with Prospect to show the runtime reduction by Prospect’s parallelization, and
4. with OOB but without Prospect to show the slowdown of the OOB without parallelization.

All runs without Prospect are single-threaded, whereas with Prospect we make full use of all 8 cores. The Prospect overhead (configuration 2) is most visible for Whetstone (3.1x) and LinPack (2.8x).
In Figure 6 we plotted the speedup of OOB with Prospect (i.e., configuration 3) relative to OOB without Prospect (i.e., configuration 4). To estimate the maximal possible speedup on our 8 core machine, we use the following upper bound:

\[
\text{slowdown} = \frac{\text{runtime configuration } 4}{\text{runtime configuration } 1}
\]

\[
\text{upper bound} = \frac{\text{number of cores}}{1 + \frac{1}{\text{slowdown}}}
\]

This upper bound takes into account that the fast variant needs about \(\frac{1}{\text{slowdown}}\) of the CPU cycles of the slow variant.

Figure 6 only presents the speedups for exactly 8 cores. In [Stüßkraut et al. 2009] we also measured the scalability of our out-of-bounds checker with Prospect with a lesser number of cores to evaluate the scalability of Prospect. Our results are that Prospect scales linearly for a low number of cores. The scalability is limited by a saturation point below the theoretical upper bound. These results are confirmed by Figure 6.

5.1.2 FastAssert

We tested FastAssert with real world code. We choose the words unit-test of BOOST’s multi-map implementation [Munoz 2009]. Figure 7 (left) shows the runtime of the test in three configurations:

1. without assertions and without Prospect,
2. with assertions but without Prospect, and
3. with assertions and with Prospect (FastAssert).

Configuration 1 is 458x faster than configuration 2. This is an unusual runtime overhead for assertions. FastAssert is 7.2x faster than configuration 2 (right hand side). But the runtime overhead compared to configuration 1 is still impractical 64x. Nevertheless, we believe that given more cores FastAssert would reduce the runtime of configuration 3 even more. We believe this example shows that FastAssert enables the inclusion of heavy-weight user defined sanity checks into production code.

5.2 Prospect Overhead

To analyze the runtime overheads introduced by Prospect, we measured the overhead of system call speculation, deterministic replay and the StackLifter individually. Figure 8 shows the overhead of system call speculation and deterministic replay for the Vacation benchmark with four different workloads (number of performed transactions). System call speculation and deterministic replay are implemented by our Linux Kernel module. We run Vacation only with the kernel module. The slow variant was forked right before the execution of the main function. Slow and fast variant share the same code. To avoid measuring overhead of the StackLifter, the whole execution took place within one epoch. No further instrumentation (especially no StackLifter) was applied. The overhead of the smaller workloads is dominated by the start-up time of our kernel module. For the two larger workloads the overhead is around 2.5%.

Figure 9 shows some overheads introduced by the StackLifter. Again, we executed Vacation with four different workloads. The StackLifter adds instrumentation to both, the fast variant and the slow variant. In this experiment, we measured the overheads introduced by this instrumentation but not the stack lifting process itself. We run both variants separately without system call speculation and deterministic replay. Applying StackLifter to all code of the fast variant increases the runtime up to 1.8x compared to vacation without Prospect. If StackLifter is restricted to functions on the path to prospect_chkpt, the overhead is below 3%. The instrumentation of the slow variant introduces a higher overhead. Instrumenting all functions, the overhead is between 2.41x and 4.60x. Instrumenting only functions on the path to prospect_chkpt, the StackLifter’s overhead is reduced to between 1.80x and 3.0x. The overhead of the slow variant can be further reduced by optimizing indirect function calls. Note that all function pointers (also in the slow variant) point to functions of the fast variant. Hence, before each indirect call, we need to look up the function pointer of the slow variant. A map is indexed by fast variant function pointers (see Listing 3). The last optimization is to add a one element look-up cache to fp2cp. This optimization is not used in the fast variant, therefore, it does not influence the overhead of the fast variant. The slow variant’s overhead is reduce to 1.16x for the largest tested workload.
In our experience, the time needed to switch from the fast variant to the slow variant is very small. Hence, it is difficult to get reliable measurements from our benchmarks. Therefore, we built a micro-benchmark executing a recursive function and calling prospect_chkpt exactly once. Besides StackLifter, we did not apply any other instrumentation. Figure 10 shows the time needed to switch from the fast variant to the slow variant for growing stack depths. Each stack frame contains one label and three live integer registers. Unsurprisingly, it takes longer to un- and rewind from greater stack depths. A linear relation exist, indicating predictable behavior.

The runtime overhead in the slow variant is noticeable for large stack depths. However, this runtime overhead is already parallelized by Prospect.

6. Related Work

The predictor/executor approach was first introduced in the hardware community [Zilles and Sohi 2002]. A distilled program (i.e., a fast variant) is generated at compile time. The execution is similar to our approach. The main difference is on how the switch from the fast variant to the slow variant at epoch boundaries is performed. [Zilles and Sohi 2002] use a hash-map to translate program counters. States are communicated via a check-pointing unit in the memory subsystem. However, it is not clear how this approach handles different stack layouts between fast and slow variant.

FastTrack [Kelsey et al. 2009] is the system most similar to Prospect. In contrast to Prospect, FastTrack is not designed to apply the approach to the whole application. It only supports fast-track regions, which must not contain system calls. FastTrack does not need a StackLifter nor does it need speculative system calls. The other way around, we see that the StackLifter is a crucial part to apply the approach to the whole application. Additionally, FastTrack makes no use of speculation for state, added to the slow variant.

In the last few years, the predictor/executor approach has got some attention from the runtime checking community [Nightingale et al. 2008; Ruwase et al. 2008; Wallace and Hazelwood 2007]. All these projects make use of dynamic binary instrumentation with the help of the Pin tool [Luk et al. 2005]. Whereas Pin allows adding code like runtime checks, it is not suitable for efficiently removing code like user-defined assertions. A StackLifter is not needed as the state of the runtime checks are completely separated from the application state. SuperPin [Wallace and Hazelwood 2007] does not support speculation for system calls. Hence, speculative state might become visible due to unsafe optimizations or failing runtime checks. Speck’s [Nightingale et al. 2008] support for system call speculation and deterministic replay is closest to ours. It is derived from Speculator [Nightingale et al. 2005]. Speculator is an operating system extension and it supports the speculative execution of one process. The speculation is propagated throughout the system, whereas, Prospect provides isolation. Hence, it is possible to run several applications under Prospect at the same time. DIFT [Ruwase et al. 2008] uses a non-trivial hardware extension to stream data from the core, running the fast version to the slave cores. The slave cores use this data to perform runtime checks.

The StackLifter solves the on-stack replacement problem that also occurs in the dynamic software update problem. The current version of an application shall be replaced by a new version without terminating or restarting the application [Fink and Qian 2003; Makris and Bazzi 2009; Neamtiu et al. 2006]. Previous work, like Ginseng [Neamtiu et al. 2006], avoids stack rewriting by only upgrading a function that has currently no frames on the stack. Data is accessed indirectly to allow online updates. The recently published UpStare [Makris and Bazzi 2009] uses an approach similar to our StackLifter. However, UpStare works on C source code and seems to need manual intervention for mapping an old version’s stack frame to a new version’s stack frame, e.g., when pointers are involved. We avoid this issue by using the alloca stack and with the help of the state equivalence property. On-stack replacement (OSR) for the JVM [Fink and Qian 2003] is quite similar to the StackLifter. The goal of [Fink and Qian 2003] is to switch from the current code base once to another, perhaps more optimized, code base. Because the switch happens once for a code base pair, the new code base is specially prepared. The current state of the stack is explicitly inline into the new code base as the starting state for each function. Furthermore, OSR works on Java bytecode, which is more abstract than the LLVM bytecode.

7. Conclusion

Prospect facilitates the implementation of parallelized and scalable runtime checkers. New checkers are relatively easy to program. In contrast to previous work, we even permit the parallelization of application programmer defined sanity checks. Prospect provides good speedups of instrumented programs and a low overhead compared to the sequential execution of an uninstrumented program.

We plan to extend Prospect in various ways. Currently, Prospect uses fixed-size epochs. We expect to achieve even better scalability when switching to dynamic epoch lengths. Prospect aborts the execution on a mispeculation. This is actually desirable for runtime checking and user defined sanity checks because a mispeculation always means that a check has failed. We plan to add a recovery mechanism in which an executor can spawn a new predictor in case of a mispeculation. We will therefore need to implement a “bi-directional” StackLifter, i.e., one that can also switch from the slow to the fast variant.

References


