Parallelizing Mudflap Using Thread-Level Speculation on a Chip Multiprocessor

Sanghoon Lee    James Tuck
Center for Efficient, Scalable, and Reliable Computing
Department of Electrical & Computer Engineering
North Carolina State University
{shlee5, jtuck}@ncsu.edu

Abstract

Pointer-use checking is an important technology for detecting pointer and array access errors in unsafe languages like C and C++. However, the large run-time overheads associated with these techniques restrict their usefulness to software development environments. On the other hand, if the execution time overhead from using these techniques can be reduced, not only can they be applied more aggressively during development, but they may also be employed in production settings.

Leveraging concurrency between the application and pointer-use checking codes is one possible way to accelerate these systems. The purpose of our study is to study and evaluate the parallelization techniques needed to support concurrent, frequent, fine-grained pointer-use checking code on CMPs. To carry out this study, we selected Mudflap, a widely distributed pointer-use checking tool integrated into GCC. We have analyzed the implementation of Mudflap and identified Thread-Level Speculation as a promising approach for parallelization. We have implemented a parallelized version of Mudflap as an extension to the POSH compiler.

1 Introduction

Software bugs are a major problem facing industry. The National Institute of Standards & Technology estimated in 2002 that inadequate infrastructure for testing software cost close to $22 billion [23]. The same study predicts that if only half of the bugs that remained in post-production software were detected and eliminated, that cost could be recouped. Consequently, techniques that increase the likelihood of finding and fixing software bugs are of significant importance.

One important class of bugs that are often reported as causing a large number of errors are pointer and array access errors [1, 6, 15, 16, 17, 19, 20]. This class of errors stems from languages like C and C++ which allow unchecked access to memory through pointers and arrays. Staticaly, such errors are tough to find due to the limitations of pointer analysis [9], and at runtime, errors are often detected long after the erroneous memory access has occurred. Consequently, to catch pointer and array access errors, a program’s execution is typically monitored in detail to detect the first instance of a read or write to an invalid memory location. However, these checks are costly, often lengthening execution time by an order of magnitude. For example, Memcheck [20] is known to slow the execution of some programs by as much as 30 times. Purify [17], a lightweight tool in comparison, often adds non-trivial cost on the order of 3 to 5 times slower. These overheads likely limit the frequency of use. Furthermore, if the execution time overhead from using these techniques can be reduced, not only can they be applied more aggressively during development, but they may also be employed in production runs.

One option for speeding-up error checking code is concurrent execution between the checking code and the application [13, 15]. While previous work ([13]) has shown that speculative threads on a CMP can support fine-grained concurrency with tolerable overheads, there has been less attention to the characteristics of such software or how to effectively parallelize it for this purpose.

To gain insights into the parallelization of these codes, we selected Mudflap[6], a widely distributed pointer-use checking tool integrated into GCC [24]. Mudflap works by annotating every pointer and array access in a program to check for a variety of memory errors including NULL pointer dereferencing, running off the ends of buffers, and memory leaks. To exploit concurrency between Mudflap operations and the application, a suitable concurrency model must be selected. We have analyzed the implementation of Mudflap and identified Thread-Level Speculation as a promising approach for parallelization. We have implemented a parallelized version of Mudflap as an extension to the POSH [12] compiler that is targeted
to a CMP with support for Thread-Level Speculation (TLS) [21, 22]. The contributions of this work are:

- Characterization of the performance and behavior of a widely used tool for detecting pointer-based bugs.
- Exploration of Mudflap’s potential for parallelization using Thread-Level Speculation.
- Discussion of the challenges that the parallelization of Mudflap poses that, as of yet, do not have effective solutions.

2 Background

2.1 Reducing Overhead of Pointer-use Checking

Many recent works have focused on pointer-use checking to detect memory errors [1, 4, 15, 16, 17, 20]. Some of these have focused on reducing the overheads of pointer-use checking. Patil and Fischer [15] focused on parallelizing checking of pointer and array accesses on multiprocessors. However, due to the large overheads of communication and synchronization on their target systems, their approach was to replicate all code that generated pointer dereferences and execute it in a separate process. For work that could not be replicated, like program inputs or outputs, the two processes synchronized. They were able to achieve concurrent execution of pointer-use checking with low execution time overhead, but they replicated a large fraction of the program’s execution on an additional processor. In our approach, this overhead can be eliminated using fine-grained threading only on the checking code.

Perhaps most related to this work is the work of Oplinger and Lam [13] that investigates the use of speculative threads for software reliability. Oplinger and Lam were the first to point out the value of using speculative threads to hide the overheads of data monitoring and checking code, like the kind Mudflap uses for pointer-use checking. They considered monitoring code in general and investigated techniques for optimizing it to run in parallel with the application. However, this work primarily described architectural mechanisms to support speculation for this purpose, and some simple techniques to eliminate common sources of data dependences. In this paper, we argue that these techniques are insufficient to effectively parallelize Mudflap.

2.2 Pointer-use Checking and Mudflap

Mudflap [6] is a set of passes and libraries integrated into GCC [24] to facilitate pointer use checking. The Mudflap pass in GCC transparently adds error checking code at the dereferencing of any pointer to validate the memory access against a database of allowed memory regions in the stack, heap, static variable section, and others. GCC provides a library, called libmudflap, that implements functions to populate the database with newly allocated memory regions (_mf_register), to remove items from the database (_mf_unregister), and to assert that a dereference is valid (_mf_check). If _mf_check is called on a region of memory that was never registered using _mf_register or which has been unregistered using _mf_unregister, a violation is detected and reported back to the user.

Libmudflap also provides wrapper functions for many C library functions. Since the C library is not compiled with Mudflap annotations, library functions that manipulate pointers must be handled as a special case. For the case of malloc and free, a wrapper function registers or unregisters, respectively, the allocated and freed region. Figure 1 illustrates how a C program might be annotated using Mudflap.

```c
int *x; int size = 2;
...
x=WRAP_MALLOC(sizeof(int)*size);
  /* the real malloc is wrapped to register the allocated region with libmudflap */
...
__mf_check(x+0,4);
x[0]=3;
__mf_check(x+2,4); /* Error! */
x[2]=3;
free(x); /* unregister x */
...
mf_check(x+1,4); /* Error! */
...=x[1];
```

Figure 1: Example code with Mudflap annotation.

2.3 Thread-Level Speculation

A TLS compiler breaks hard-to-analyze sequential code into tasks, and speculatively executes them in parallel, hoping not to violate sequential semantics (e.g. [2, 3, 5, 10, 27, 28, 29]). The control flow of the sequential code imposes a control dependence relation between the tasks. This relation establishes an order of the tasks, and the terms predecessor and successor express this order. The sequential code also yields a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

A task is speculative when it may perform operations that violate data or control dependences with its predecessor tasks. When a non-speculative task finishes execution, it is ready to commit. The role of commit is to inform the rest of the system that the data generated by the task are now part of the safe, non-speculative program state. Among other opera-
tions, committing always involves passing the non-speculative status to a successor task. Tasks must commit in strict order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires non-speculative status.

As tasks execute in parallel, the system must identify any cross-task data dependence violation. Typically, this is done with special hardware support that tracks the data read and written for each task. A data dependence violation is flagged when a task modifies a version of a datum that was read prematurely by a successor task. At this point, the successor is squashed and all the state that it has produced is discarded, and its successor tasks are also squashed. Then, the task is re-executed.

TLS architectures can discard the state produced by a task and re-start the task thanks to special hardware that buffers all speculative modifications, and a checkpointing mechanism that enables rollback [7, 11, 21, 22, 25, 26].

3 Mudflap Characterization

The execution characteristics of Mudflap that determine how well it can be parallelized are dependent both on its high level algorithm and on the specific implementation that has been selected. Since the point of this paper is not to re-design Mudflap, we are primarily interested in those parts of the algorithm which may enable or prevent parallelization. We dissect the discussion into parts: first, we discuss the parallelization opportunities present in the high level workings of Mudflap; next, we consider the optimizations already adopted in Mudflap to achieve reasonable performance.

3.1 High-level Behavior

There is good reason to expect that Mudflap has a significant amount of parallelism. The interface to Mudflap is made up of just a few functions. The most common ones are _mf_check, _mf_register, and _mf_unregister. These functions are used to check, add, and remove entries to a data structure that keeps track of valid memory regions. In a correct program, _mf_register will be called on a region of memory before any call to _mf_check or _mf_unregister. Also, _mf_unregister will be called after all calls to _mf_check for that same memory region. Clearly, calls to _mf_register and _mf_unregister will update the data structure, potentially creating a dependence between itself and other updates.

-mf_check looks for a valid entry, and based on this fact alone, there is no inherent reason that _mf_check should update the data structure or the entry it’s checking. (However, there are other reasons it may do so as described in the next section.) Therefore, separate calls to _mf_check, even to the same memory object may be able to proceed in parallel. Since calls to _mf_check are, essentially, as common as a dereference in the source code, such calls will be abundant and hopefully can be parallelized.

3.2 Serial Optimizations

Mudflap performance will be highly dependent on application characteristics like the frequency of pointer manipulation, the kinds of data structures used (linked data structures or not), the size of the working set, and other aspects of the annotated code [6]. These factors will influence the amount of annotation and the execution length of each check. This may vary significantly with a program’s phases or inputs. To help mitigate some of these factors, Mudflap has been carefully designed to minimize its impact on sequential execution.

Leverages existing compiler optimizations. The Mudflap instrumentations are performed only after all other optimizations in GCC’s frontend. This includes optimizations like dead code elimination, partial redundancy elimination, and dead store elimination.

Uses a lookup cache for frequent checks. Programs are likely to access the same memory address close together in time. A lookup cache that memo-
izes the result of the last check can prevent expensive calls to _mf_check. Mudflap implements a direct mapped software cache with a configurable number of entries, usually around 1000. Each entry in the cache stores a contiguous range of valid addresses, or an empty range if it has not been used.

GCC generates in-lined code at each unique address calculation that is eventually used to reference memory. The in-lined code calculates an index (bit-wise mask and shift) to the lookup cache and tests if the entry stored there is valid for the address. If the entry is valid, no other work is needed. If the entry is not valid, _mf_check is called and the cache is updated to reflect the largest range of contiguous addresses it is contained within. As a result of this implementation, accesses to the same cache entry but different regions of memory have a serial dependence.

Adapts the lookup cache hashing function. Instead of using a common hash function for the lookup cache for all applications, Mudflap adapts the hashing function based on the regions of memory being used. However, in order to calculate this hash function, each call to _mf_check updates statistics about the object in memory: in particular, how many times it is referenced. This information is used to compute
Table 1: Characterization of Mudflap with respect to Baseline system. 

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>7.9</td>
<td>11.2</td>
<td>51.2</td>
<td>1988.9</td>
<td>14.4</td>
<td>25.5</td>
<td>1.1</td>
<td>1.5</td>
<td>3.0</td>
<td>0.8</td>
<td>98.6</td>
<td>97.4</td>
</tr>
<tr>
<td>crafty</td>
<td>14.4</td>
<td>15.6</td>
<td>82.0</td>
<td>2017.7</td>
<td>4.6</td>
<td>7.0</td>
<td>1.2</td>
<td>1.3</td>
<td>2.3</td>
<td>0.7</td>
<td>93.0</td>
<td>94.0</td>
</tr>
<tr>
<td>gap</td>
<td>3.3</td>
<td>4.8</td>
<td>21.6</td>
<td>1792.3</td>
<td>7.0</td>
<td>50.6</td>
<td>1.0</td>
<td>1.5</td>
<td>1.3</td>
<td>0.4</td>
<td>94.2</td>
<td>96.0</td>
</tr>
<tr>
<td>gzip</td>
<td>5.5</td>
<td>8.6</td>
<td>75.1</td>
<td>2068.1</td>
<td>0.0</td>
<td>13.3</td>
<td>0.9</td>
<td>1.4</td>
<td>6.9</td>
<td>1.4</td>
<td>92.6</td>
<td>96.0</td>
</tr>
<tr>
<td>mcf</td>
<td>4.4</td>
<td>7.6</td>
<td>50.4</td>
<td>1942.7</td>
<td>0.0</td>
<td>35.5</td>
<td>0.8</td>
<td>1.3</td>
<td>24.4</td>
<td>7.1</td>
<td>88.4</td>
<td>94.6</td>
</tr>
<tr>
<td>parser</td>
<td>3.9</td>
<td>5.6</td>
<td>54.1</td>
<td>2017.3</td>
<td>17.5</td>
<td>30.5</td>
<td>1.0</td>
<td>1.5</td>
<td>11.6</td>
<td>1.9</td>
<td>96.1</td>
<td>96.4</td>
</tr>
<tr>
<td>twolf</td>
<td>20.8</td>
<td>26.5</td>
<td>86.0</td>
<td>2464.2</td>
<td>1.9</td>
<td>8.5</td>
<td>0.9</td>
<td>1.2</td>
<td>0.5</td>
<td>1.5</td>
<td>89.3</td>
<td>92.9</td>
</tr>
<tr>
<td>vpr</td>
<td>43.4</td>
<td>56.4</td>
<td>92.7</td>
<td>2477.4</td>
<td>2.0</td>
<td>3.5</td>
<td>1.0</td>
<td>1.2</td>
<td>2.8</td>
<td>0.3</td>
<td>89.5</td>
<td>93.4</td>
</tr>
<tr>
<td>Geo.Mean.</td>
<td>8.7</td>
<td>12.1</td>
<td>55.8</td>
<td>2084.0</td>
<td>0.0</td>
<td>13.8</td>
<td>1.0</td>
<td>1.3</td>
<td>3.5</td>
<td>1.1</td>
<td>92.6</td>
<td>95.1</td>
</tr>
</tbody>
</table>

Table 1: Characterization of Mudflap with respect to Baseline system. 

a weighted distribution over the bits in the address space. Updating these statistics on each _mf_check call adds a dependence between checks to the same object.

Uses a splay for efficient data organization. To efficiently record all live objects as well as represent their memory range concisely, a splay is used. A splay is advantageous for performance since it amortizes the cost of balancing the tree and recently accessed items are quick to access again. The use of a splay is highly effective for the temporal locality present in a program’s reference stream.

However, each search in a splay reorganizes the splay to move the searched item, or it’s nearest neighbor, to the root of the tree. As a result, each splay operation creates a serial dependence with splay operations that follow. Since _mf_check is the most common operation and will operate on the splay frequently, the splay introduces many serial dependences between all consecutive calls to _mf_check, even when the checks are for objects in different regions of memory.

While the fundamental mudflap algorithm is parallel in principle, these optimizations are designed to boost single thread performance and introduce many serial dependences between Mudflap operations that make parallelization challenging.

3.3 Characterization

Table 1 shows a comparison of several SPECint applications running with and without Mudflap. The experimental setup for the runs is shown in Table 3.

The first column shows the application name. The second column shows the execution time of Mudflap normalized to the baseline. The additional overhead is up to 43 times in the case of vpr, and over the entire suite a geometric mean of 9 times slower. The normalized instruction count, shown in the third column, is somewhat higher ratio than the execution time.

The next three columns explain where the sources of overhead come from. Column Check shows the fraction of instructions that directly comes from _mf_check. The next column shows the average size in instructions of each call to _mf_check. Column Other is the overhead from all other explicit calls into the library, like _mf_register and _mf_unregister. Finally, Column Hash is an estimate for the in-lined cache lookup overhead. However, it also includes any overhead that is caused from less optimized code from GCC’s backend as a result of Mudflap instrumentation. Of the three columns, Check is the greatest contributor to Mudflap’s overhead with Hash roughly half as important on average. The addition of in-lined monitoring contributes approximately 16% of the dynamic instructions in a Mudflap execution. However, individual applications show a variety of behavior, with crafty, twolf, and vpr showing more than 80% of instructions in calls to _mf_check.

The last three pairs of columns compare the instructions per cycle (IPC), the L1 Data cache miss percentage, and the branch predictor (BP) accuracy percentage for Base and Mudflap. Note that IPC increases across the board. L1 Data cache miss rates decrease for Mudflap except for twolf. Overall, branch predictor accuracy increases. While many instructions have been added, they are relatively independent and are well behaved.

4 Parallelizing Mudflap

Our overall goal is to accelerate Mudflap on a Chip Multiprocessor. Based on the characterization in the previous section, the performance bottlenecks of Mudflap that need to be resolved are (1) the cost of in-lining many checks to the lookup cache, and (2) the frequent calls to _mf_check. We will put off the discussion of how to handle (1) to Future Work (Section 7). In this section, we focus on parallelizing frequent calls to _mf_check.

4.1 Why Thread Level Speculation?

Many models of parallel execution may be suitable like TLS, Transactional Memory [8], or Decoupled Software Pipelining [14]. For this work, we have cho-
Thread-Level Speculation for the following reasons.

- Mudflap is a tool for program debugging. As a result, preserving sequential semantics is critical for usability. When a Mudflap violation occurs, TLS can provide a snapshot of the program state at the time of the violation. While this may not always be helpful, it may be better than providing a partially updated state.

- As shown in Table 1, the size of each call to \_\_mf_check is small. Furthermore, we know there are many dependences between consecutive calls to \_\_mf_check even when different memory objects are being checked. Thread-Level Speculation is better at handling this situation since partial results from one thread can be pipelined to more speculative threads regardless of when and where the two calls to \_\_mf_check occur.

- The serial optimizations that are used by Mudflap are important both for serial performance and parallel performance. Even though the lookup cache introduces dependences it saves many calls to \_\_mf_check. Hence, the serial code optimizations cannot be discarded—instead we should embrace them to make parallelization more efficient as well. Thread Level Speculation can be applied efficiently in the context of frequent cross-thread dependences, as long as they can be anticipated in advance.

- Finally, since the calls to \_\_mf_check are a result of the dynamic properties of the lookup cache, parallelism expands and shrinks in highly irregular patterns. It is not centric to loops (the ideal target for DSWP), rather it is dispersed dynamically throughout the control flow graph. Thread Level Speculation is well suited for these irregular parallelization patterns.

### 4.2 Parallelization

As described in Section 3.3, \_\_mf_check has several limiting dependences. Figure 2 shows a simplified data dependence graph for \_\_mf_check. To parallelize \_\_mf_check we must cope with these dependences using a known strategy, like (i) rewriting the algorithm to eliminate them, (ii) synchronizing them when possible, (iii) speculating that in the common case they will not occur, or (iv) allowing a race which may lead to misspeculation.

**Eliminating or coping with Splay induced dependences.** By virtue of their design, operations on splay trees often induce true dependences from one search to the next. This dependence is labeled (1) in Figure 2. We consider two mechanisms for handling this dependence.

If an application is not sensitive to the benefits of a splay (i.e., quicker subsequent searches for nearby items), then replacing it with a simple binary search tree will eliminate dependences without affecting overall performance. Also, since the splay is implemented as an abstract data type, its implementation for search can be replaced easily without disrupting the workings of Mudflap.

However, if replacing the splay search operation increases the amount of work required over the run of the whole application, it may be difficult to recoup the lost work using parallelization. For some applications, we observed a loss of performance of up to 30% when moving from a splay to a binary search tree. For these cases, synchronization may work better.

**Lookup cache synchronization & speculation.** The lookup cache is necessary for performance, since it saves a significant volume of work. However, it adds a serial dependence chain to all \_\_mf_check calls that use the same lookup cache entry. The dependences caused by the lookup cache are labeled (2) in Figure 2. Note that one edge in the graph is a self dependence late in the function, and the other is an edge.
spanning from a late write to an early load. These edges will only be true dependences when either two calls to `_mf_check` access the same cache entry, or when the first in-lined lookup misses and the second one should read the first’s update.

Unfortunately, since read-modify-write sequences are common in programs, the long dependence edge from a later store to an early load will occur. Since it would be wasteful to duplicate the work just to avoid a dependence when accessing the lookup cache, the second cache lookup should serialize with the first on a cache miss. If such sequences are not too common, synchronization is a simple and effective solution.

However, if this dependence is common, synchronization of this dependence is less than desirable because it is written late in one thread and read at the beginning of the other. This forces the in-lined lookup to wait for a previous call to `_mf_check` to complete. Since the in-lined code is not in a parallel task, the application will be stalled waiting for the `_mf_check` to proceed.

To avoid this overhead, value speculation may be used to break the dependence. Instead of waiting until the end of the function to update the cache entry, it is updated immediately upon entering `_mf_check`. Many different value prediction schemes are possible with varying degrees of accuracy. However, a particularly interesting value predictor can be derived from the Mudflap algorithm: rather than predict the maximum range of a valid object to store in the cache, just store the bounds of the current object being checked in the cache. As long as the object is valid, it is a conservative solution, and it is available without additional computation on entry to the `_mf_check`. This allows the value prediction to occur as soon as the lookup cache miss is detected. A value misspeculation would only be flagged when the object turns out to be invalid. Note, however, that this value prediction scheme may change the hit rate of the lookup cache and hurt the overall efficiency of Mudflap.

Statistics collection for cache adaptation. For the most part, statistics collection for cache adaptation can be left as a data race or eliminated depending on the parallelization model. In general, it is not necessary to have a perfect record of object reference counts, and some applications are not as sensitive to cache adaptation as others. If collection is mandatory for a given purpose, they can be synchronized.

Table 2 summarizes the key dependences limiting parallelization and the strategies adopted for overcoming them. The last column identifies the experiment configurations evaluated in Section 6 for each strategy.

<table>
<thead>
<tr>
<th>No.</th>
<th>Strategy</th>
<th>Exp. Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sync. all splay searches</td>
<td>Mflap+TLS</td>
</tr>
<tr>
<td>2</td>
<td>Re-write without splay search</td>
<td>(Any)+Splay.</td>
</tr>
<tr>
<td>3</td>
<td>Sync. lookup with update</td>
<td>Mflap+TLS</td>
</tr>
<tr>
<td></td>
<td>Predict cache update early</td>
<td>(Any)+VP.</td>
</tr>
<tr>
<td>4</td>
<td>Synchronize (Dep. 2 covers it)</td>
<td>(Any)w/out VP, Mflap-Adapt</td>
</tr>
</tbody>
</table>

Table 2: Strategies for dealing with the dependences in `_mf_check`.

5 Implementation

5.1 Compiler Support

The parallelization of Mudflap requires two components: placing all calls to `_mf_check` generated by GCC into a TLS task, and parallelization of intercepted library calls (like malloc, free, and other pointer-centric manipulation routines like strcpy). We leverage TLS support from the POSH [12] compiler (upgraded to GCC 4.3) to generate code for TLS tasks.

Since we must manually parallelize some C library calls that Mudflap patches, we extended POSH with limited support for manual parallelization using pragmas. Using GCC’s multilib support in conjunction with the pragmas, we can build both parallelized and non-parallelized versions of the supporting Mudflap libraries. Figure 3 shows how the pragma is used to parallelize a library call patched by Mudflap.

For the strategies discussed in Section 4, we manually implemented all cases in libmudflap.

```c
void * WRAPPED_memmem (const void *haystack, size_t haystacklen, const void *needle, size_t needlelen)
{   TRACE (%s
    #pragma posh task
}   MF_VALIDATE_EXTENT(haystack, haystacklen, __MF_CHECK_READ, "memmem haystack");
    MF_VALIDATE_EXTENT(needle, needlelen, __MF_CHECK_READ, "memmem needle");
}   return memmem(haystack, haystacklen, needle, needlelen);
}
```

Figure 3: Parallelization of Mudflap function wrapper for memmem using POSH pragma.

5.2 Architectural Mechanisms

Scheduling Our TLS parallelization strategy aims to preserve the application’s critical path of execution by placing calls to `_mf_check` into parallel tasks that run on other cores. Figure 4 illustrates this pro-
Figure 4: A comparison of an application with no Mudflap annotation (far left), Mudflap annotation with a serial execution, and Mudflap annotation executing in TLS tasks.

Depending on the availability of resources, the forked tasks may execute in parallel on multiple cores (B and C) or sequentially on a single core (A and B). We assume that TLS hardware will track the task ordering sequence (1,A,2,B,3,C,4) regardless of which core the task executes on.

Synchronization To synchronize accesses to the splay and to the lookup cache, we assume a signal and wait mechanism in hardware. The wait command takes a flag value as an argument and forces a thread to stall until it sees a signal on the same flag. A safe thread will never stall on a wait command. Also, we assume that a signal operation can only be received by successors in the TLS program order.

6 Evaluation

6.1 Setup

We have evaluated our compiler pass and manual parallelization of Mudflap on SESC [18], an event-driven performance simulator with a detailed model of TLS execution. Table 3 shows the simulated architecture.

6.2 Applications

The application configurations are shown in Table 4. Per the discussion in Section 4.2, we created a binary for each parallelization of interest. These applications include + TLS in their name to indicate the presence of overhead associated with TLS task generation and to indicate they are linked with the parallelized version of libmudflap. We also created a few different versions of Mudflap for serial execution (Mflap+VP, Mflap-Adapt, Mflap-Splay (described in Table 2)) to evaluate the impact of various algorithm changes outside of the context of parallelization.

To accurately compare the performance of different binaries, simply timing a fixed number of instructions cannot be used. Instead, “simulation markers” are inserted in the code of each binary, and simulations are run for a given number of markers. After skipping the initialization (typically 1-6 billion instructions), a certain number of markers are executed, so that the baseline binary graduates from 10 to 100 million instructions in the Baseline binary. These simulation lengths are small due to the large overhead added by Mudflap.

6.3 Parallelization Results

Figure 5 shows the speedup over Mflap for the four parallelized implementations of Mudflap. While the average results are low, a handful of applications did achieve a significant performance boost, namely, crafty, twolf, and vpr. These applications have the largest fractions of their dynamic execution devoted to handling calls to _mf_check. Since our parallelization strategy primarily focused on _mf_check, it is not surprising they benefited the most.

Overall, configurations with value prediction achieved a small advantage over the others, with a geometric mean of 1.06. This is likely because it
relaxed some synchronization constraints that freed tasks that were actually independent. However, there is no overall trend suggesting one parallelization strategy worked well across the board. If we collect the best speedups for each application, the geometric mean jumps up to 1.19.

To further explain some of the behavior seen in the speedups, we evaluated the effect of the algorithmic changes on the serial execution of the program. The results are shown in Figure 6 as speedup over Mflap. Note that Mflap+VP implements the value prediction heuristic as the permanent policy for updating the Cache. This has a large impact on performance, in some cases a factor of 5 slowdown. This explains why twolf, which is very sensitive to this policy, shows little benefit from value prediction, while crafty gets a significant boost in Figure 5. Also, the inefficiency of this heuristic serves as a reminder how poor performance can be without an effective lookup cache.

Mflap-Adapt does not collect statistics regarding object usage. Applications which are sensitive to the Cache hash function suffer the most from eliminating these statistics—namely, bzip2 and gzip. Since the gains from parallelization are small, adapting the Cache hash function is more important.

Finally, Mflap-Splay shows the impact of replacing the Splay search with a binary search. Overall, the performance loss from this policy is small. However, the loss is significant for twolf and crafty. This is why Mflap+TLS-Splay performs worse for these two applications.

Given the large slowdowns caused by Mudflap, the meager performance gains achieved here are somewhat disappointing. However, they are not a result of pathological TLS behavior. On average, squash rates ranged from 0.5% on average for Mflap+TLS and gzip. Since the gains from parallelization are small, adapting the Cache hash function is more important.

7 Summary & Future Work

Since detecting bugs is critical to the reliability and security of software systems, we are examining techniques to parallelize pointer-use checking on CMPs. By making these checks more efficient, they can be used in a wider variety of settings including post-development environments. To carry out this study, we have implemented a parallelized version of Mudflap as an extension to the POSH [12] compiler that is targeted to a CMP with support for Thread-Level Speculation (TLS). Our implementation almost doubled the performance for one application, and, taking the best parallelized version for each application provided a 19% speedup on average over the fully serial algorithm.

Future Work. While previous work has focused on using speculative threads to boost various error checking codes, there has been less focus on the mechanisms needed to effectively parallelize rule-based bug detection schemes. In this article, we applied many of the basic mechanisms known to boost performance under TLS: synchronization, value speculation, and algorithm re-writing. However, achieving good performance for a wide range of applications remains elusive.

To further boost performance, dependences that serialize main application behavior need to be eliminated. Of the dependences shown in Figure 2, the serializing dependence on the cache is the most problematic. Because the in-lined lookup to the cache is not within a speculative task, it may force the main application to stall while it synchronizes with a predecessor updating the same cache entry. If it is allowed to serialize calls to \_mf\_check too much, then little parallelization can be gained. This appears to be the key performance limiter for many of the applications. However, as discussed in Section 4, if it were eliminated entirely, it would increase Mudflap overheads dramatically.

Furthermore, our approach failed to consider the large overheads added by direct in-lining of code. Since the trend is toward more, simpler cores on a CMP, it is unlikely that that ILP alone can compensate for this additional code. Therefore, the in-lined code must also be parallelized. Due to the limitations of POSH, simply placing all in-lined code into tasks is also not an effective solution. Since POSH uses register spilling to communicate values between threads, the overhead from creating a speculative task at all memory dereferences significantly diminishes the IPC of the parallel program. Hence we need to investigate strategies that further eliminate serializing dependences by grouping together many checks into a single thread, especially ones that are likely to be dependent. Such a strategy will require significant involvement from the compiler since this code is a product of the main application. These challenges highlight the need to look deeply at the techniques required to effectively parallelize pointer-use checking codes and error checking codes in general.

References


Figure 5: Speedups normalized to Mflap.

Figure 6: The impact of parallelism enhancing techniques on the serial program.