# Automatic Parallelization of Fine-Grained Meta-Functions on a Chip Multiprocessor[*]

Sanghoon Lee James Tuck
*Department of Electrical & Computer Engineering*
*North Carolina State University*
{shlee5, jtuck}@ncsu.edu

## Abstract

*Due to the importance of reliability and security, prior studies have proposed inlining meta-functions into applications for detecting bugs and security vulnerabilities. However, because these software techniques add frequent, fine-grained instrumentation to programs, they often incur large runtime overheads. In this work, we consider an automatic thread extraction technique for removing these fine-grained checks from a main application and scheduling them on helper threads. In this way, we can leverage the resources available on a CMP to reduce the latency and overhead of fine-grained checking codes.*

*Our parallelization strategy automatically extracts meta-functions from the main application and executes them in* customized *helper threads — threads constructed to mirror relevant fragments of the main program's behavior in order to keep communication and overhead low. To get good performance, we consider optimizations that reduce communication and balance work among many threads.*

*We evaluate our parallelization strategy on Mudflap, a pointer-use checking tool in GCC. To show the benefits of our technique, we compare it to a manually parallelized version of Mudflap. We run our experiments on an architectural simulator with support for fast queueing operations. On a subset of SPECint 2000, our automatically parallelized code is only 29% slower, on average, than the manually parallelized version on a simulated 8-core system. Furthermore, two applications achieve better speedups using our algorithms than with the manual approach. Also, our approach introduces very little overhead in the main program — it is kept under 100%, which is more than a 5.3× reduction compared to serial Mudflap.*

## 1. Introduction

One major trend in computing is the continuing increase in the complexity of software systems. Such an increase is motivated by the expectation of increasingly powerful hardware (faster processors, larger memory, etc.), programming environments that provide higher abstraction to the programmers, and the increasing diversity of environments in which the software runs. To ensure efficient, reliable, and secure execution of complex software systems, programmers increasingly rely on meta-functions, functionalities that do not contribute directly to the output of a program but help ensure efficient, reliable, and secure execution of the program. Various forms of meta-functions are implemented today in response to the diverse requirements of software execution.

At the same time, the trend in microprocessor design is toward many-core systems. The performance of a single core in a many-core system will increase at a slower speed than in the past, and in some cases will stagnate. In order to exploit the performance of such a system, programmers need to exploit parallelism at various levels in their applications.

Software's increasing reliance on metafunctions present a promising source of parallelism. The predominate approach in performing metafunctions is to interleave their execution with the main application code, through direct augmentation of the metafunction code into the application code or by multiplexing the processor at run-time between the application and its metafunction execution. Such an interleaved execution approach invariably increases the critical path of the main application, sometimes increasing it by orders of magnitudes. For example, Memcheck, a tool that runs using Valgrind, has been reported to slow down applications by a factor of 2× to 30× [19, 36, 31], and even then it only checks for a subset of memory errors.

Only recently has parallelism been exploited for performing metafunctions, such as memory management [13, 35], and bug detection [32, 20]. However, these techniques mainly focused on long-running, coarse-grained meta-functions.

In this work, we focus on automatic extraction of fine-grained meta-functions for parallel execution on multi-cores. Our goal is the development of a compiler which can efficiently extract meta-function level parallelism from programs in a way that provides low latency execution and is scalable on multi-core systems available today or in the near future. One of the primary challenges for fine-grained meta-functions are the abundant instructions inserted through-

[*]This work was supported in part by NSF Award CNS-0834664.

out the main application. Reducing this fine-grained interleaved overhead is difficult to accomplish while still providing scalability. Our strategy is the generation of customized helpers which follow the call context of the main program and replicate as much state as possible in a way that does not require communication. Using a smart task partitioning strategy, meta-function code and some of its dependences are migrated to the customized helper. This approach results in low overhead in the main thread since meta-functions and their support code need not be executed there. To provide scalability, we generate multiple customized helpers using static load-balancing heuristics.

In our work, we make the following contributions:

- We propose a novel parallelization methodology using Custom Helpers for fine-grained meta-function parallelization.
- To provide scalability, we describe simple heuristics to generate multiple helpers and balance their load.
- We evaluate our framework on Mudflap, a pointer-use checking tool implemented in GCC. We run our experiments on an architectural simulator modeling an aggressive 8-core CMP with support for fast queueing operations. We show that our automatic technique is efficient and scalable; it achieves an average 1.4× speedup using 8 cores. Compared to a manually optimized version of the Mudflap framework, our approach is only 29% slower. Furthermore, our custom helpers reduce the overhead in the main thread by 5.3× compared to serial Mudflap.

The remainder of the paper is organized as follows: Section 2 describes the key ideas behind our compiler and parallelization strategy; Section 3 presents our compiler algorithms; Section 4 explains how Mudflap is mapped into our framework; the experimental setup and evaluation are in Section 5; related work is given in Section 6. Section 7 concludes.

## 2. Fine Grained Meta-function Parallelization

Meta-functions are extra functionality added to a program to augment its behavior in some useful way. While they can be an arbitrary function, we restrict their definition, somewhat, to cover common uses and to limit the scope of our study. We assume that meta-functions have the following properties. (i) They are placed at a program point, $p$, to compute some desired property at that location. (ii) They take as input State$_p$ that is extracted from the main program state and MetaState$_{p_{in}}$ that is information calculated specifically by the meta-functions. A critical simplifying assumption is that MetaState is disjoint from State. (iii) They produce as output an optional error state, Error$_p$, and optional updates to the meta-function state, MetaState$_{p_{out}}$. Note that none of these meta-functions update State; we do not allow meta-functions to update program state in order to narrow the scope of this work.

**Table 1. Characterization of Mudflap overheads normalized to an optimized baseline without it.**

| App | Norm Exe | Norm Inst. | % of Dyn. Inst. Check | Other | Hash | Chk Avg |
|---|---|---|---|---|---|---|
| bzip2 | 11.1 | 19.2 | 51.8 | 14.2 | 25.3 | 1988.2 |
| gap | 6.6 | 9.7 | 21.1 | 5.4 | 53.6 | 1646.0 |
| gzip | 3.9 | 6.2 | 52.0 | 0.0 | 26.7 | 2011.6 |
| mcf | 5.6 | 15.7 | 55.1 | 0.0 | 34.1 | 1945.4 |
| parser | 7.1 | 11.0 | 15.9 | 22.3 | 39.7 | 1871.3 |
| twolf | 6.1 | 10.0 | 12.7 | 12.3 | 53.3 | 1664.5 |
| vpr | 5.4 | 8.2 | 25.6 | 22.7 | 33.1 | 1994.7 |
| Geo.Mean. | 6.2 | 10.7 | 28.7 | 13.7 | 36.5 | 1868.7 |

We will assume that most meta-functions fall into two categories: *validating meta-functions* and *updating meta-functions*. A *validating meta-function* checks a property of MetaState$_{p_{in}}$ without modifying it and determines if State$_p$ contains an error. This is essentially a complex check of some dynamically invariant property. An *updating meta-function* uses State$_p$ to transition MetaState$_{p_{in}}$ to MetaState$_{p_{out}}$ and may detect an error related to the update operation. For example, in pointer-use checking, an important *validating meta-function* is testing that a dereference occurs to memory that has been allocated and not freed; an example of an *updating meta-function* is tracking when an object is allocated or freed so that the database of known memory objects is kept up-to-date.

There are several costs associated with inserting meta-functions into a program. First and foremost, the meta-function must be executed as part of the main program, thereby adding considerable runtime overhead. This instruction payload can increase runtime by orders of magnitude. Second, the inlining of code and function calls worsen the compiler's ability to optimize the main program. As a result, the efficiency of the main application is much worse than it would be without meta-function instrumentation. The first two effects are clearly discernible from direct inspection of the generated code. Thirdly, the additional code competes with the main program code for critical processor resources, like issue slots, space in the cache memory hierarchy, or space in branch predictor history tables.

Table 1 shows some key overheads from Mudflap, a pointer-use checking tool. All percentages are shown relative to a binary without any Mudflap support. This data was collected using the same experimental setup described in Section 5. The fraction of updating and validating meta-functions combined is approximately 42% (Check + Other); this fraction represents time spent in the computational region of meta-functions. Additional inlined-code makes up another 36% (Hash) of the program. The total increase in instructions, on average, is 10.7×.

A naive way to parallelize meta-function workloads is by re-writing the meta-functions to explicitly leverage parallelism. Such a meta-function level approach to parallelization is straightforward to achieve. In such a parallelization strategy, each meta-function would decide the best way to parallelize it's task. For example, it may send all of its inputs to waiting helper threads that are ready to receive and process them. At a minimum, it will need to send State$_p$. With frequent meta-function operations, the main thread will spend considerable time forwarding tasks to helpers. This approach can achieve good performance, but it requires significant programmer effort and cannot exploit redundancy or eliminate inefficiency that may exist across meta-functions within the same program.

### 2.1. Our Parallelization Approach: Custom Helpers

In our approach, we want to extract meta-function code out of the main program and run it in parallel. By automatically extracting meta-function code to helper threads, we can construct highly tuned, custom helpers that match the main program's needs. Looking at each input to a meta-function provides some insight on how to do this. Consider State$_p$. The same state may be needed by multiple meta-functions. Through code analysis, the compiler can generate a *custom helper* which receives the input value once and uses it in each meta-function call. Furthermore, rather than communicating just the inputs to meta-functions, main can communicate values for a code slice that the helper uses to generate the full State$_p$ for one or more meta-function calls. By sending a only a few inputs for a slice, some work in the main thread can be eliminated and communication with the helper can be reduced. Location$_p$ can also be handled more efficiently. If a custom helper is generated for a specific region of code, then some inputs can be embedded directly in the helper without any need for communication.

In summary, our compiler will generate a custom helper thread that is highly aware of the main program. It would be possible to generate a custom helper for the entire program (essentially a reduced program clone) or just for parts of it. So that we do not need to choose regions for parallelization, we generate a custom helper for the entire main program. The helper will mimic the behavior of the main program by matching its call stack dynamically. Relationships among multiple meta-functions will be exploited through a task selection algorithm that aggressively moves or replicates code from the main thread into the custom helper.

### 2.2. Providing Scalability

Given the large computational overhead introduced through meta-functions, multiple threads are needed to successfully parallelize them. For example, as shown in Table 2, there is no hope of significantly reducing the overhead in *bzip2* with only one helper thread. Ideally, performance should scale as additional resources are added; once enough resources are added, the impact on the main program should be minimal. Therefore, we support multiple custom helpers.

In order to support multiple threads, meta-function parallelism must be exposed to the compiler. A general way of expressing and operating on such parallelism is beyond the scope of this paper. Currently, we make decisions about which meta-functions can run in parallel at compiler compile-time to narrow the scope of the paper. In other words, they are fixed for a given set of meta-functions. To decide which meta-functions can run in parallel, we leveraged profiling information and our understanding of the meta-function implementation as much as possible[1].

To provide scalability, multiple custom helpers can be generated for the same function. Each custom thread will receive part of the meta-function workload of a function using a static load balancing algorithm. However, load balance is challenging using a static algorithm for a variety of reasons. First, meta-functions may have dynamically varying execution times. For example, in order to reduce the time of meta-function execution, memoizaiton is often used to significantly reduce overheads. As a result, the same meta-function can have wildly varying execution times. Second, the execution path of the program is unknown and can only be estimated. To properly balance load, you want the meta-functions that will actually execute to be distributed evenly. Finally, the structure of some program regions may not lend themselves to parallelization over many threads. For example, an inner loop may have one meta-function. If the loop is not unrolled and the trip count is high, then all executions of the inner loop body will be scheduled on the same helper resulting in imbalance. Analogous behaviors occur with many different code structures.

To compensate for all of these effects with a static algorithm is complex and perhaps intractable. Therefore, for this study, we adopt simple techniques and show that they work reasonably well. The two heuristics we consider are random assignment and round robin.

### 2.3. Summary

In summary, our overall parallelization approach aims to reduce the overhead in the main application by extracting customized helpers that provide low overhead meta-function execution. However, to combat the high overheads of meta-function's themselves, we leverage multiple helpers to scale up to many threads and keep the overhead in the main program low. We will show that this approach is competitive with manual parallelization in our case study.

## 3. Parallelization Algorithm

We have designed a compiler algorithm that extracts and parallelizes meta-functions. In the explanation that follows, we start with the parallelization of a single function that includes meta-functions. Then, we expand it to support generation of the base helper thread which mimics the main thread's behavior. Next, we describe how to use the base helper thread parallelization algorithm for custom helper thread extraction. Finally, we describe how to scale the algorithm to multiple threads using static load balancing techniques.

[1]This is non-trivial to perform automatically, but the implementer of such meta-functions can determine parallelization properties with reasonable effort.

## 3.1. Helper Function Parallelization Algorithm

**Algorithm 1**: CreateHelper

```
   Input: f: original function
   Input: P: set of statements to move to helper thread
   Input: R: set of statements to replicate in helper thread
   Input: tid: thread id
   Result: Customized helper function, $helper$, and modified f as a side-effect.
1  build PDG for f   /*CDG + DDG*/ ;
2  $S = P \cup R$ ;
3  $A = \emptyset$ ;
4  foreach $s \in S$ do
5      $A = A \cup \{m \mid m$ is an ancestor of $s$ in CDG subgraph $\}$ ;
6  end
7  $S = S \cup A$ ;
8  $helper, map = Clone(f, S, tid)$   /* create clone of $f$ restricted to $S$ */ ;
9  foreach stmt $s' \in helper$ do
10     foreach use $u \in s'$ do
11         $MaybeInsertCommChannel\,(u, f, helper, map)$ ;
12     end
13 end
14 remove $P$ from $f$ ;
15 return $helper$
```

We base our parallelization algorithm for a single function on the Program Dependence Graph (PDG) [7]. We include all control, register and memory dependence edges in our PDG. The $CreateHelper$ algorithm in Algorithm 1 shows the algorithm for creating the helper function.

The inputs to $CreateHelper$ are the original function in SSA form ($f$), a subset of statements $P$ to include only in the helper, and a subset of statements $R$ to include in both. We assume that extracting $P$ and $R$ to the helper will not result in cyclic communication between the main program and the helper thread; rather, communication flows in a single direction from the main thread to the helper. $CreateHelper$ assumes that $P$ and $R$ have been selected properly, so no such check is repeated here.

In order to move a statement to the helper, a few actions are required. First, we build a PDG [7] (a unified control and data dependence graph) for $f$. We set $S = P \cup R$ so that we have all statements to be included in the helper in one place. Next,all control predicates for each statement in $S$ must be included in the helper (lines 4-7). Once all the nodes in the helper are computed, a clone of the function can be created with only the selected statements and control predicates present (line 8). For each input or predicate value, a communication channel, in the form of a queue, is added between the function and the helper (lines 9-12). In line 14, instructions in $P$ are removed from $f$.

*MaybeInsertCommChannel* has some smarts built into it that are not shown here. First, we check if $u$ really needs to be communicated: is $def(u)$ in $f$ but not in $helper$? This test covers three cases. Either the definition has been moved to $helper$, it has been replicated in $helper$, or it is only in $f$. We only communicate $u$ if it is not available in $helper$. If we do need to communicate $u$, we proceed as follows. Since $f$ is in SSA form, we insert a $send(u)$ just after its definition in $f$, and we replace its definition in $helper$ with $u = recv()$. We also keep a record that $u$ was communicated and never insert another channel for it between $f$ and $helper$.

One additional detail worth noting. *Clone* forces the new helper function to have a signature with no input parameters and no return value. While this is not strictly necessary, it simplifies our implementation since all helpers will have no inputs or outputs.

## 3.2. Baseline Helper Extraction

**Algorithm 2**: BaseHelperThread

```
   Input: $f$: original function in SSA form
   Input: $P$: set of nodes to move into helper
   Input: $R$: set of nodes to replicate in helper
   Input: $tid$: custom thread id
1  foreach $stmts \in f$ do
2      if s is call stmt then
3          $R = R \cup s$ ;
4      end
5  end
6  $f' = CreateHelper(f, P, R, tid)$ ;
7  if f is externally visible then
8      insert 'send(f)' at entry of $f$;
9  end
10 return $f'$;
```

Our baseline helper is a clone of the entire main program that mimics the main application's call stack. This ensures the helper is very lean and efficient by default. Whenever the main application calls a function the helper should call and execute an associated helper version.

The algorithm is shown in several parts. First, in the $BaseHelperThread$ function (Algorithm 2), we accumulate all call statements into the $R$ set for parallelization. Next, we generate the helper with those statements extracted using the $CreateHelper$ algorithm. $CreateHelper$ works as previously described but we can now consider more of the details in the $Clone$ function, shown in Algorithm 3. $Clone$ will copy the function exactly, however, it will remove all statements that are not in $S$, and it will replace all call statements that are not meta-functions with a code sequence that will call their associated helper.

**Algorithm 3**: Clone

```
   Input: $f$: function in SSA form
   Input: $S$: set of statements to be included in clone
   Input: $tid$: custom thread id
1  $f'$ = clone of $f$   /* exact replica of f but with a new name */ ;
2  $map = CreateMap(f, f')$ ;
3  foreach stmt $s \in f$ do
4      $s' = map[s]$ ;
5      if s not in S then
6          remove $s'$ from $f'$;
7      end
8      if s is a call && not a meta-function then
9          $ReplaceCall(s, s', tid)$ ;
10     end
11 end
12 $CHHT[f, tid] = f'$;
13 remove parameters and make void type ;
14 return $f'$ ;
```

Replacing the calls turns out to be more involved than it may appear at first. There are several cases which must be supported for this to work for arbitrary programs. First, indirect function calls through pointers can only be resolved at runtime. Therefore, the mechanism for determining which helper to run must occur at runtime (at least some of the time). Second, functions in libraries may not have a corresponding helper. Ideally, this should be prevented by providing a wrapper function for the library call that makes it compatible with our system, but this is not always possible. So, the external library call will not have a corresponding helper, but it could still call yet another function which does have an associated helper. The `qsort` function is a good example; a function pointer is passed as an argument and is used to help sort the data structure. Such a function passed as an argument is now *externally visible*. The `qsort` function may not have a helper, but its function argument likely will. While neither case is very common, both do occur with regularity in applications and must be supported.

**Algorithm 4**: CHHTLookup

```
   Input: $funptr$: function pointer to callee in main
   Input: $tid$: custom thread id
1  if funptr then
2      if funptr == 1 then
3          $funptr = recv()$ ;
4      end
5      $newptr = CHHT[funptr, tid]$;
6      $newptr()$ ;
7  else
8      repeat
9          $funptr = recv()$ ;
10         if funptr then
11             $newptr = CHHT[funptr, tid]$;
12             $newptr()$ ;
13         end
14     until funptr ;
15 end
```

**Algorithm 5**: ReplaceCall

```
   Input: $s$: original stmt in f
   Input: $s'$: cloned stmt in f'
   Input: $tid$: custom thread id
   Result: Replaces callee in $s'$ with call to a helper function
1  $callee$ = Callee($s$) ;
2  if $callee$ is external then
3      replace $s'$ with call to 'CHHTLookup(0,$tid$)' ;
4      insert 'send(0)' after $s$ ;
5  else
6      if $callee$ is externally visible then
7          replace $s'$ with call to 'CHHTLookup(1,$tid$)' ;
8      else
9          replace $s'$ with call to 'CHHTLookup(callee,tid)' ;
10     end
11 end
```

**Custom Helper Hash Table.** In order to dynamically bind a function with its customized helper, a Custom Helper Hash Table (CHHT) is constructed which creates a mapping between a function and its helper [2]. Since we may generate multiple custom helper threads, it also needs a thread id as input.

The $CHHTLookup$ function shown in Algorithm 4 is used at runtime to find the dynamic mapping of function to its custom helper. There are several cases that are required for correctness. We will consider each case in turn.

**Direct and Indirect Function Calls.** To support direct and indirect calls, at each call site we must determine the mapping from callee to helper thread. We can accomplish this with the $CHHTLookup$ function by passing the address of the callee as an argument. In the case of the indirect call, the computation of the callee's address must be sent to the helper as a result of lines 9-12 in $CreateHelper$. For direct calls, the callee's address can be encoded directly in the helper. The relevant lines of code are 5-6 in Algorithm 4, and 9 in Algorithm 5.

**Externally Visible Calls.** Since we cannot modify all of the call sites of externally visible functions, we must communicate with the helper from inside the function rather than at the call site. At such a call site that we can analyze, we replace the call with $CHHTLookup$ with a value of 1 as $funptr$ (6-7 in Algorithm 5). This will force $CHHTLookup$ to wait until it receives one function pointer (line 2) from the called function; then it uses it to perform the lookup and call the corresponding helper function.

When $BaseHelperThread$ analyzes the externally visible function, it inserts the 'send(f)' code (lines 7-9 in Algorithm 2).

**External Calls.** If an externally defined function (like `sort`) is called and there is no wrapper function available, then it may call externally visible functions which do have helpers. In this case, we replace the external call with a call to $CHHTLookup$ with a 0 as its $funptr$ argument. This will force the helper to wait until it receives a function pointer from an externally visible function (lines 7-14 in Algorithm 4). However, because it may never call such a function or it may call such functions repeatedly, we must identify when $CHHTLookup$ terminates by sending a null pointer after the external call ends (2-4 in Algorithm 5).

**Optimized Call Sequence.** This mechanism works for all function calls, but it is sometimes inefficient. With additional information about the function, these costs can be avoided. For example, in the case of a static function in C, we need not call $CHHTLookup$ at runtime to determine the mapping. Instead, we can hardcode the helper function's location directly into its caller and allow function inlining to further optimize the code. We do not show it in our algorithm, but we do take this additional step in our full implementation.

## 3.3. Custom Helper Extraction

In Custom Helper Extraction, meta-functions and their support code are extracted from a function in the main thread and scheduled in the customized helper thread. We assume that a function, $f$, is already annotated with all meta-functions and supporting code, and we assume that meta-function calls can be identified and inserted into a set, $M$, of statement nodes. Using only $M$ as the basis for helper extraction would yield an inefficient extraction with all meta-function inputs communicated explicitly to the helper. Instead, we take a smarter approach and look for additional statements to include in $M$ that will reduce the communication overhead between $f$ and the helper.

Our algorithm is shown in Algorithm 6. Using SSA

[2] To create the $CHHT$, the compiler builds a sequence of code to be executed at the beginning of the program that initializes the $CHHT$ and inserts all known mappings.

```
void foo()
{
    int x, y;
    x = 1;
    /* Static function call  */
    x = goo(x);
    /* Predicate node  */
    if(x==1)
    {
        /* Stmts for meta-functions */
         y = x + 3;
         meta_check(y, __FILE__, __LINE__);
     }
}
```

(a) Original version

```
void foo()
{
    int x, y;
    /* Stmt only in Main */
    x = 1;
    x = goo(x);
    send(x);



}
```

(b) Main thread version

```
void foo.helper()
{
    int x, y;
    /* Replace static function call */
    CHHTLookup(goo,1);
    /* Communication for data  */
    x = recv()
    if(x ==1)
    {
        y = x +3 ;
        meta_check(y, __FILE__, __LINE__);
     }
}
```

(c) Helper thread version (tid is 1)

**Figure 1. A simple example of our Helper Extraction.**

**Algorithm 6**: CustomHelper

```
Input: $f$: function in SSA form
Input: $M$: set of meta-function nodes to move into helper
Input: $tid$: custom thread id
Result: Custom helper function
repeat
    foreach $s \in M$ do
        foreach $use\ u \in s$ do
            if all uses of u are in M then
                $M = M \cup def(u)$ ;
            end
        end
    end
until $M$ converges ;
return $BaseHelperThread(f, M, \emptyset, tid)$ ;
```

form, we efficiently iterate over every statement $s$ in $M$. For each use $u$ in $s$, if $u$ is only used by statements in $M$, then the definition of $u$ can be added to $M$ as long as it adds no extra communication (or even reduces it). The extraction phase finishes once all members of $M$, both new ones and ones added along the way, have been considered. This algorithm will see to it that any data only computed for the meta-function is included in $M$. Finally, we use the $CreateHelper$ function to parallelize the statements in $M$ by setting the input argument $P = M$. Note, this algorithm will never create cyclic communication between $f$ and the helper.

Interestingly, a valuable benefit of this technique is the extraction of context sensitive constant values. For example, many meta-functions take code location as an argument. For a given custom helper, code location amounts to a constant value which will easily be extracted by the algorithm since it is only used in the meta-function.

### 3.4. Example

Figure 1 shows a simple example of our Helper Extraction. Figure 1(a) shows the original function. Using our algorithms, we generate functions for the main thread (Figure 1(b)) and the helper thread (Figure 1(c)), respectively. This example illustrates how our algorithm replaces a call, communicates a predicate node, and places the computation of some inputs only in the helper thread. Statement *"x = goo(x)"* is a direct call which must be preserved in the main thread and replaced in the helper with $CHHTLookup$, as shown in Fig. 1(c). Next, consider the predicate node with the expression *"(x==1)"*. Since a meta-function depends on this predicate, it must be preserved, and our algorithm is smart enough to communicate x and compute *"(x==1)"* in the helper. Note that the main thread sends the predicate value using *"send(x)"* and the helper thread receives the predicate value using *"x=recv()"*. Both operations occur at the same program point as the original definition of *x*. Finally, operations that support constant inputs of meta-functions are only placed in the helper. *__FILE__ and __LINE__* are two such examples. However, since *y* is an input, we must communicate *x* in order to compute *y*. In this case, our algorithm is smart enough not to send *x* twice.

### 3.5. Supporting Multiple Helpers

To generate $N$ custom helpers, we use a load balancing algorithm to create $N$ unique partitions from the $M$ set. Then, for each tid and $M_{tid}$ from 1 to $N$, we use the $CustomHelper$ algorithm to generate a custom helper.

We have experimented with two static load balancing techniques to create the $N$ partitions of $M$: round robin and random. For our round robin approach, we traverse the CFG in DFS order and assign meta-functions to a partition using round robin. This is a simple approach that we have found to work well. We also considered random assignment. We use the same basic algorithm as round robin but place meta-functions into threads randomly. Random assignment can avoid imbalance caused by systematic interactions between the code structure and round robin assignment. More sophisticated approaches are possible, but we have found that these are competitive in our current implementation.

## 4. Case Study: Mudflap

Mudflap [5] is a set of passes and libraries integrated into GCC [34] 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.

In this section, we consider how to support Mudflap parallelization in two ways. We will discuss how Mudflap maps into our automatic parallelization framework and discuss some synergistic benefits we discovered. Also, we will describe our strategy for manually parallelizing it. In the evaluation, we will compare these two approaches.

**Table 2. Meta-functions used by Mudflap.**

| Meta-function Name | Inputs | Insertion Point | Description |
|---|---|---|---|
| `__mf_register` | Pointer to buffer, size, location | Any memory allocation point | Record allocated memory region in a database |
| `__mf_unregister` | Pointer to buffer, location | Any memory deallocation point | Remove previously allocated region from database |
| `__mf_check` | Pointer, ref. size, location | All memory references | Validate reference is for a registered region of memory |
| inlined software cache lookup | Pointer, size | All memory references | Checks a software cache to see if reference has already been validated |

## 4.1. Mapping Mudflap to our Framework

Table 2 shows the meta-functions used by the Mudflap algorithm. `__mf_register` is an *updating meta-function* added at every memory allocation point to record a new memory region that can be accessed by the program. Such memory allocation points include malloc, stack allocation, and global variables. `__mf_unregister` is an *updating meta-function* that is inserted at all deallocation points. And, `__mf_check` is a *validating meta-function* that checks if a memory reference falls within an allocated region. Figure 2 provides an example of a program annotated with Mudflap's meta-functions.

We have studied the behavior of Mudflap to determine the appropriate strategy for parallelizing these meta-functions. Since `__mf_register` and `__mf_unregister` are relatively infrequent and since the database they create is relatively small, they are always replicated across all helpers. `__mf_check`, on the other hand, is very frequent and must be balanced across the helpers to provide scalable performance improvement. Table 1 from Section 3 backs up these decisions. GCC provides a library, called *libmudflap*, that implements these metafunctions. We modified this library to support a parallel implementation. In particular, we changed the code so that a database could be created and maintained per thread.

Since the C library is not compiled with Mudflap annotations, library functions that manipulate pointers must be handled as a special case. Libmudflap already provides wrapper functions for many libC functions that allocate memory. For the case of `malloc` and `free`, a wrapper function performs the register or unregister, respectively, on the allocated or freed region. We modified all such wrapper functions manually to provide a customized helper and added them in the CHHT.

**Benefits for Mudflap.** Our framework is expected to provide several useful benefits for Mudflap. Looking at the inputs for each metafunction, it is clear that two of the inputs are likely constants: size and location. These inputs will be extracted directly into the custom helper. The pointer must be communicated, but if it is used in multiple meta-functions it may only need to be communicated once. Finally, the inlined software cache lookup is directly inserted into the program to prevent redundant calls to `__mf_check`. These inlined codes add significant instruction and control flow overhead and all of these can be moved to the helper.

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

**Figure 2. Example code with Mudflap annotation.**

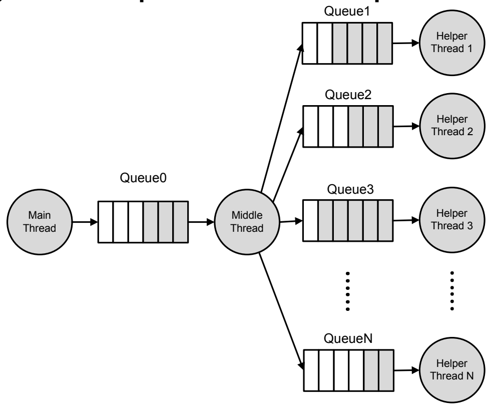

**Figure 3. Mudflap Manual Parallelization.**

Our framework also provided some unexpected benefits. Local aggregates need to be registered with Mudflap. However, when a helper is created, we initially copy all local variable declarations into the helper. Since the helper's copy is not the same memory location as the one in the main program, the local copy in the helper need not be registered with Mudflap. But, we found it beneficial to do so. By registering the helper's local aggregate, many meta-functions could be executed using the local aggregate without ever receiving input from the original function. An out-of-bounds check in the helper on a copy is equivalent to a check in the original function. This optimization reduced communication and, thereby, lightened the load on the main program. (N.B. We still register the address of the original aggregate from the main program just in case its address is ever taken.)

### 4.2. Manual Parallelization

In order to evaluate how well our automatic technique works, we also manually parallelized Mudflap for comparison. In this implementation, we just re-wrote the meta-functions to exploit parallelism explicitly.

The meta-functions implement a pipelined architecture as shown in Figure 3. For each meta-function in Mudflap, we created a parallel version which packs up its input and enqueues them to Queue0. Each meta-function inserts a special header at the beginning of its input so that it can be parsed properly by the Middle Thread. Also, we modified `__mf_check` to include the software cache lookup. This modification significantly reduces the code injected into the main program. Also, to make `__mf_check` a little more efficient, *we omitted the debugging location information*. For every pointer use check, we insert this alternative implementation of `__mf_check`.

The Middle Thread parses the contents of Queue0 and decides how to handle the meta-function. It must send `__mf_register` and `__mf_unregister` requests to all helpers. For `__mf_check` requests, it applies a load balancing heuristic and sends the request to one of the Helper Threads for completion. Because the performance of the system is very sensitive to the processing rate of the Middle Thread, the load balancing heuristics are kept simple and fast. We considered two dynamic load balancing techniques: address-based balancing and round robin.

**Parallelization 1: Address indexing.** Because the Helpers use a software cache to filter out redundant checks, we prefer to send redundant checks to the same core so they will be filtered quickly. We use a simple hash function on the input address to select a queue.

**Parallelization 2: Round-robin.** We also evaluate the Middle thread using a round-robin method to select a helper thread. Using this approach, allocation is more uniform, and the Middle thread is less likely to block on a full queue.

Our round robin algorithm sends 8 consecutive operations to the same Helper. By sending 8 at a time, we exploit locality in the access stream; consecutive checks are more likely to refer to the same object in the Mudflap database. We determined a chunk size of 8 through experimentation.

## 5. Evaluation

### 5.1. Setup

We have evaluated our design on SESC [10], an event-driven performance simulator, that accurately models CMP architectures. Table 3 shows the pertinent details of the simulated architecture. The core, L1, and L2 cache are common for all architectures we evaluated. The core is an aggressive 3-issue superscalar processor. We model a cache coherent CMP design with private L1 and L2 caches. Coherence is supported at the L2 cache over a ring interconnect.

For this study, we use highly efficient hardware queues for communication between threads. We assume 1 cycle to read/write from/to the queue. Also, we used 4 KB for each hardware buffer. We used hardware queues rather than software queues to showcase the effectiveness of our parallelization strategy.

Table 4 shows the different application binaries we study. For all of our application binaries, we used GCC 4.5 targeted to the Mips32 architecture. All of our compiler algorithms described in this paper were implemented as a plugin for GCC. 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, a certain number of markers are executed so that the binary graduates from 100 to 500 million instructions.

**Table 3. Architecture simulated. All cycle counts are in processor cycles.**

| 8 Core CMP | |
|---|---|
| Frequency 3.2 GHz<br>Fetch width 6<br>Issue width 3<br>Retire width 3<br>Branch predictor:<br>Mispred. Penalty 14 cycles<br>BTB 2K, 2-way | ROB 126<br>I-window 68<br>LD/ST queue 48/42<br>Mem/Int/Fp unit 1/2/1<br>Hybrid GShare & Local |
| Private L1 Cache:<br>Size 64KB, assoc 4<br>Line 64B<br>Latency 2 cycles | Private L2 Cache:<br>Size 2MB, assoc 8<br>Line 64B<br>Latency 9 cycles<br>Protocol MESI<br>Memory:<br>Latency 300 cycles<br>Bandwidth 25.6GB/s |

**Table 4. Configurations and parallelizations.**

| Name | Description |
|---|---|
| Base | No Mudflap,-O3 |
| Mflap | Base + Mudflap |
| A-Random | Automatic Parallelization with Random Assignment |
| A-RoundRobin | Automatic Parallelization with Round-robin Assignment |
| M-Index | Manual Parallelization with Address Indexing Assignment |
| M-RoundRobin | Manual Parallelization with Round-robin Assignment |
| BHE4 | Baseline Helper Extraction with 4 threads |
| BHE8 | Baseline Helper Extraction with 8 threads |

### 5.2. Baseline Helper Extraction

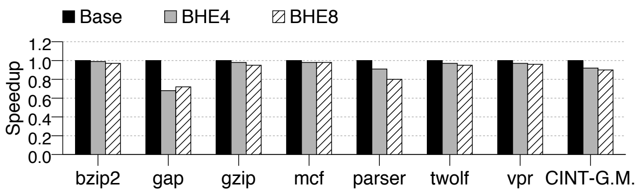

**Figure 4. Scalability of Base Helper Extraction with 4-cores and 8-cores.**

In this section, we evaluate the scalability of our $BaseHelperThread$ extraction algorithm with several

SPECint applications on a 4-core and 8-core CMP. Figure 4 shows the speedup over Base. CINT-G.M represents geometric mean normalized to Base over these applications. On average, our proposal performs 8% worse than Base for both 4-core and 8-core CMP configurations. However, *gap* loses 30% and *parser* loses 20% while the others only lose between 2∼5%. That is, Helper extraction is scalable for 8-cores with only 2∼5% loss. *gap* and *parser* show a larger slowdown compared to the others because they have a large number of function calls. The calls themselves alone are not expensive, but in order to match the control path of the main thread, they require frequent communication of predicates which add a lot of overhead to main. With this overhead, *gap* and *parser* are limited in the parallelism they can support.

## 5.3. Custom Helper Extraction

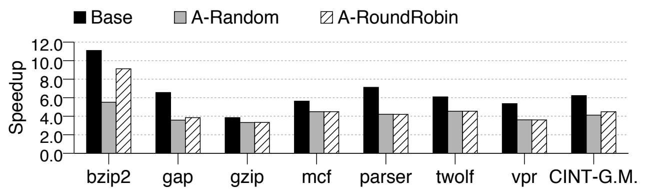

**Figure 5. Speedups of Parallelized Mudflap with empty run-time functions.**

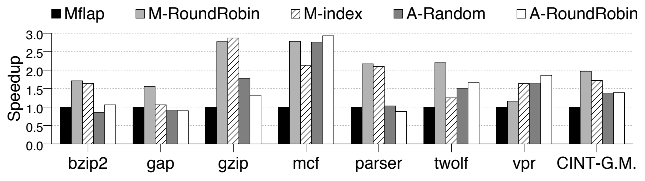

**Figure 6. Speedups of Parallelized Mudflap.**

In this section, we evaluate several characteristics of the parallelized Mudflap meta-functions of several SPECint applications on the 8-core CMP. Figure 5 shows the speedup over Mflap. For this evaluation, we use empty `__mf_register`, `__mf_unregister` and `__mf_check` meta-functions to show the quality of parallelization without the effects of runtime load imbalance. In addition, we compare our two static algorithms for load balancing. Looking at the overall geometric mean, our design achieves 4.1× speedup with random assignment and 4.5× speedup with round-robin assignment. This suggests that our extracted helpers are quite efficient. Furthermore, the performance has only been degraded by 28% over the results in Figure 4.

Figure 6 shows the speedups compared to Mflap running the full Mudflap implementation using both the automatic parallelization and the manual parallelization. Overall, looking at the geometric mean, the automatically parallelized code is only 29% slower than the best manual technique. Furthermore, the automatic technique is performing competitively; vpr and mcf achieve their highest performance using A-RoundRobin. mcf achieves its advantage because its workload is relatively balanced, and it performs well even using a static load balancing approach. vpr has modest gains, comparatively; but, both automatic techniques are better than the manual strategies with dynamic load balancing. This case highlights the importance of automatic meta-function extraction. twolf also performs reasonably well. bzip2, gap, gzip, and parser are under performers. gzip suffers from poor load balance; the manual cases achieve speedups within 25% of Base because of their aggressive load balancing strategies. bzip2 also suffers primarily from poor load balance. Finally, gap and parser perform poorly due to a combination of factors: poor load balance and the significant communication overhead needed to support function calls.

## 5.4. Overheads

We have also measured some key overheads in our system. Figure 7 shows the relative size of the binary for three configurations: Base, Mflap, and A-RoundRobin. On average, Mflap is almost 1.71× bigger than Base. Our technique adds additional overhead due to Custom Helper creation. This overhead is 42%, on average.

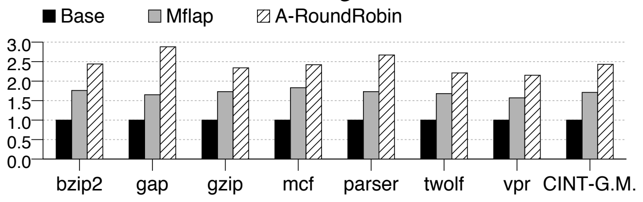

**Figure 7. Binary size.**

We also measured the number of dynamic instructions in the main thread for Base, Mflap, M-RoundRobin and A-RoundRobin. We can see that the parallelized versions do significantly reduce the overhead in the main thread. M-RoundRobin slightly edges out A-RoundRobin for two reasons: (1) the main program only communicates with one other thread which keeps its communication overhead low; and (2) we have eliminated debugging information from the `__mf_check` meta-function for manual parallelization. In contrast, A-RoundRobin must send inputs for `__mf_register` and `__mf_unregister` to all custom helpers. Furthermore, other data, like control predicates, also gets sent to multiple helpers. Currently, we do nothing to reduce this overhead, but we plan to address these communication inefficiencies in future work.

# 6. Related Work

**Helper Threads**. Many works have improved single-thread performance by using helper threads [16, 28, 35, 37, 38]. In [16, 28], to tolerate memory bottlenecks, the helper threads perform pre-execution with idle hardware resources. Xekalakis *et al.* [38] combined helper threads with Thread-Level Speculation and Runahead execution. Tiwari *et al.* [35] used a helper thread to accelerate memory management.

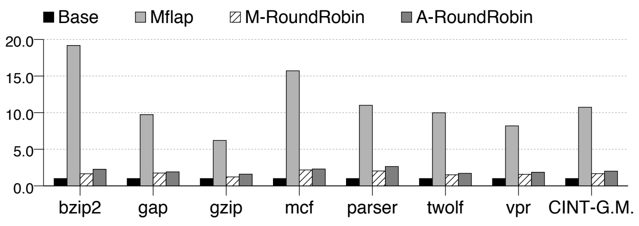

**Figure 8. The main thread's dynamic instruction counts.**

**Parallelizing Security Meta-functions**. While many security protection mechanisms have been proposed, few have investigated exploiting parallelism between the application and its security meta-functions, which is the focus of this proposal. Nightingale *et al.* proposed Speck [20], which exploits speculative parallelism between the application being protected with its security meta-functions. Shetty*et al.* [27] and Kharbutli *et al.* [13] parallelize memory management functionality for increased security. Plakal *et al.* [24] proposed parallelizing garbage collection using program slices on multiprocessors. However, they did not implement the slices, rather they only evaluated the potential.

**Thread-Level Speculation**. One alternative way to exploit parallelism between the application and its meta-functions is to use *thread-level speculation* (TLS). TLS allows frequent data sharing at low cost, but without a guarantee that the speculative work will be useful. Examples of work in bug detection with TLS include [21, 40, 12]. All studies employ thread-level speculation to hide the overhead of program checking. It has been applied in security by Speck [20], as discussed previously.

**Memory Bugs.** Detection of memory bugs also can be performed statically by utilizing explicit model checking [17, 33] and program analysis [2, 6, 8]. As software becomes more complex and diverse, users increasingly rely on run-time bug detection by instrumenting the code with checks, such as in Purify [26], Valgrind [30, 31], Intel thread checker [11], DIDUCE [9], Eraser [29], CCured [18], and others [1, 4, 15, 22, 23]. Such instrumentation typically introduces large performance overheads because instrumented memory references (loads and stores) are executed often, and execution of the instrumentation code is interleaved with normal program execution.

Hardware support for detecting memory bugs or to support debugging has been proposed, such as iWatcher [40], AccMon [39], SafeMem [25], DISE [3], and the scheme proposed by Oplinger et al. [21]. All three schemes (iWatcher, AccMon, and DISE) interleave application execution with bug checking, address matching, or debugging functions. In contrast, our helper computing mechanisms attempt to exploit parallelism between the helpers and the main application execution.

## 7. Conclusion

Due to the importance of reliability and security, prior studies have proposed inlining meta-functions into applications for detecting bugs and security vulnerabilities. However, because these software techniques add fine-grained instrumentation to programs, they often incur large runtime overheads by reducing the effectiveness of code optimization on the original program and by adding large overheads to compute the meta-function workload. In this work, we consider an automatic thread extraction technique for removing these fine-grained checks from a main application and scheduling them on helper threads. In this way, we can leverage the resources available on a CMP to reduce the latency of fine-grained checking codes.

We evaluate our parallelization strategy on Mudflap, a pointer-use checking tool in GCC. To show the benefits of our technique, we compare it to a manually parallelized version of Mudflap. We run our experiments on an architectural simulator with support for fast queueing operations. On a subset of SPECint 2000, our automatically parallelized code is only 29% slower, on average, than the manually parallelized version on a simulated 8-core system. Furthermore, two applications achieve better speedups using our algorithms than with the manual approach. Also, our approach introduces very little overhead in the main program — it is kept under 100%, which is more than a 5.3× reduction compared to serial Mudflap.

## References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[2] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[3] M. Corliss, E. Lewis, and A. Roth. Low-Overhead Interactive Debugging Using DISE. In *Proc. of the International Symposium on High Performance Computer Architecture*, 2005.

[4] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *In Proc. of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[5] F. Eigler. Mudflap: Pointer use checking for C/C++. In *Proc. of the GCC Developers Summit*, 2003.

[6] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *In Proc. of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 1987.

[8] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In

*Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[9] S. Hangal and M. S. Lam. Tracking Down Software Bugs using Automatic Anomaly Detection. In *Proc. of the International Conference on Software Engineering*, 2002.

[10] J. Renau, et al. SESC. *http://sesc.sourceforge.net*, 2004.

[11] KAI-Intel Corporation. Intel Thread Checker. URL: http://developer.intel.com/software/products/ threading/tcwin. 2004.

[12] 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.

[13] M. Kharbutli, X. Jiang, G. Venkataramani, Y. Solihin, and M. Prvulovic. Comprehensively and efficiently protecting the heap. In *Proc. of International Symposium for Programming Languages and Operating Systems*, 2006.

[14] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware Accelerated Queueing For Fine-Grained Threading on a Chip Multiprocessor. In *Proc. of the 17th International Symposium on High-Performance Computer Architecture*, 2011.

[15] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via Run-Time Type Checking. *Lecture Notes in Computer Science, vol. 2029*, 2001.

[16] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proc of the 28th International Symposium on Computer Architecture*, 2001.

[17] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[18] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proc. of the Symposium on Principles of Programming Languages*, 2002.

[19] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2005.

[20] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. *SIGPLAN Not.*, 43(3):308–318, 2008.

[21] J. Oplinger and M. Lam. Enhancing Software Reliability with Speculative Threads. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[22] H. Patil and C. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. In *Software Practice and Experience*, Jan 1997.

[23] H. Patil and C. N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *Proc. of the Second International Workshop on Automated and Algorithmic Debugging*, 1995.

[24] M. Plakal and C. N. Fischer. Concurrent garbage collection using program slices on multithreaded processors. In *Proc. of the 2000 International Symposium on Memory Management*, 2000.

[25] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proc. of the International Symposium on High Performance Computer Architecture*, 2005.

[26] Rational Software. Purify. *http://www.rational.com/products/ purify_unix/index.jsp*, 2006.

[27] Rithin Shetty, Mazen Kharbutli, Yan Solihin, and Milos Prvulovic. HeapMon: a Low Overhead, Automatic, and Programmable Memory Bug Detector. *In Proc. of IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (P=ac2)*, Oct 2004.

[28] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proc. of the 7th International Symposium on High Performance Computer Architecture*, Jan 2001.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a Dynamic Data Race Detector for Multithreaded Programs. In *ACM Transactions on Computer Systems*, 1997.

[30] J. Seward. Valgrind, An Open-Source Memory Debugger for x86-GNU/Linux. *URL:http://valgrind.kde.org*, 2004.

[31] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proc. of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[32] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. A Helper-Thread Approach to Programmable, Automatic, and Low-Overhead Memory Bug Detection. *IBM Journal of Research and Development*, 2006.

[33] U. Stern and D. L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Proc. of the Conference on Correct Hardware Design and Verification Methods*, 1995.

[34] T. G. Team. GNU Compiler Collection. URL: http://gcc.gnu.org. 2008.

[35] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin. Mmt: Exploiting fine-grained parallelism in dynamic memory management. In *Proc. of the 24th International Parallel and Distributed Processing Symposium*, 2010.

[36] Valgrind Developers. The Valgrind Quick Start Guide. *http://valgrind.org/docs/manual/quick-start.html*, 2005.

[37] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen. Helper threads via virtual multithreading. *IEEE Micro*, 24:74–82, November 2004.

[38] P. Xekalakis, N. Ioannou, and M. Cintra. Combining thread level speculation helper threads and runahead execution. In *Proc. of the 23rd international conference on Supercomputing*, 2009.

[39] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-Bsased Invariants. In *Proc. of the 37th International Symposium on MicroArchitecture)*, 2004.

[40] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. of the 31st International Symposium on Computer Architecture*, 2004.