Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs

Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade

Barcelona Supercomputing Center, C/Jordi Girona, 31, 08034 Barcelona, Spain
{paul.carpenter,alex.ramirez,eduard.ayguade}@bsc.es

Abstract. Star Superscalar is a task-based programming model. The programmer starts with an ordinary C program, and adds pragmas to mark functions as tasks, identifying their inputs and outputs. When the main thread reaches a task, an instance of the task is added to a run-time dependency graph, and later scheduled to run on a processor. Variants of Star Superscalar exist for the Cell Broadband Engine and SMPs.

Star Superscalar relies on the annotations provided by the programmer. If these are incorrect, the program may exhibit race conditions or exceptions deep inside the run-time system.

This paper introduces Starsscheck, a tool based on Valgrind, which helps debug Star Superscalar programs. Starsscheck verifies that the pragma annotations are correct, producing a warning if a task or the main thread performs an invalid access. The tool can be adapted to support similar programming models such as TPC. For most benchmarks, Starsscheck is faster than memcheck, the default Valgrind tool.

1 Introduction

There is a general need for verification and debug tools for parallel programming, to make it possible to develop reliable and correct parallel software. Debuggers are suitable for sequential and explicitly threaded applications. Data race detectors find concurrent accesses to shared memory, which may result in undefined behaviour. This paper presents Starsscheck, a tool that finds undefined behaviour in a task-based parallel program.

Star Superscalar (StarSs) is a task-based programming model. The programmer starts with a sequential C program, and adds pragma annotations to mark tasks, identifying their inputs and outputs. Execution begins in the master thread on a single processor; when it reaches a task, an instance of that task is added to a run-time dependency graph, and later executed in a worker thread. The StarSs run-time system renames and tracks arrays in a similar way to register renaming in a superscalar processor, so the master thread does not have to wait before "forwarding" an output array on to a different task. There are several variants of StarSs: CellSs [1] supports the Cell Broadband Engine (CBE) [2], and SMPSs [3] targets SMP multicores.

Figure 1 shows the *bmod* function from LU factorisation. The pragma declares the function to be a task, and specifies the direction of each array argument as

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 2–13, 2010.

```
1 #pragma css task input(row,col) inout(inner)
2 void bmod(float row[32][32],
3           float col[32][32],
4           float inner[32][32])
5 {
6   for (int i=0; i<32; i++)
7     for (int j=0; j<32; j++)
8       for (int k=0; k<32; k++)
9         inner[i][j] -= row[i][k]*col[k][j];
10 }
```

Fig. 1. Example StarSs code (*bmod* function from LU factorization)

input, *output*, or *inout* (both). A full description of the programming models can be found in the CellSs [4] and SMPSs [5] manuals.

Starsscheck checks that tasks only access memory that they are supposed to: their input and output arrays, the .text code section, and the stack at and below the function's arguments. It also checks that the main thread performs a wait before it accesses a task's output and before it writes in a task's input array. Section 2 discusses the kind of bugs that Starsscheck can find.

The analysis tool runs under Valgrind [6], a widely-used framework for binary instrumentation. The default Valgrind tool, *memcheck* [7], warns the user whenever the program's behaviour depends on invalid data; for example by conditionally jumping based on the contents of memory returned by malloc. Our approach is not specific to Valgrind. We require a binary translation tool that has some mechanism similar to Valgrind's *client requests*, which are calls from the guest program into the analysis tool. The tool could be written for Pin [8].

Any StarSs program is also a valid sequential program, since a non-StarSs compiler will ignore the pragmas. Starsscheck runs the sequential version of the program under Valgrind, and checks that, for the supplied input, the pragma annotations are correct. Starsscheck can be adapted to any programming language for which a valid sequential program can be easily derived; e.g. by ignoring pragmas or keywords. It also requires the language to supply some restrictions on the regions of memory that can be accessed by a task; there is little point if a task can freely access shared memory.

TPC (Tagged Procedure Calls) [9] also provides task based programming in C; it targets fine grain tasks, and has lower task creation and scheduling overhead. The main thread initiates tasks, which similarly have each argument labelled as *in*, *out*, or *inout*. The function that implements the task returns void, but the main thread is given a handle by the run-time system, and it must wait on this handle before it can access the contents of an array again itself or forward it to a different task. Starsscheck requires only minor changes to support TPC.

Cilk-5.3 [10] is an earlier task-based programming language based on C. Unlike StarSs and TPC, Cilk is intended for shared memory, so tasks can freely access memory, and are deterministic if they respect the Cilk shared memory model. Any task can create subtasks. The scheduler uses a *work-first* policy, which means that a (sub)task immediately starts execution on the processor that created it, and the continuation of the outer task can be stolen by another processor. Since a Cilk task

implicitly waits for all its subtasks before it completes, the dependency graph is a series-parallel DAG.

2 Common StarSs Errors

Figure 2 illustrates the main errors that can be found using Starsscheck. In subfigure (a), the function accesses memory via a pointer embedded in a structure. The programmer needs to identify which pointers can be dereferenced by each task, since the StarSs run-time needs to know the dependencies between tasks, and, on distributed memory, it needs to program DMA transfers. On CellSs, p->ptr is a pointer in the PPE's address space, but here it will be dereferenced on an SPE. Both use 32-bit pointers, and these pointers are not distinguished by the C type system.

```
#pragma css task input(p) \
                 output(y)
void a(struct t *p, int y[1])
{
    y[0] = *(p->ptr);
}
```

(a) Arbitrary memory access

```
#pragma css task output(y[n]) \
                 input(x[n], n)
void b(int *y, int *x, int n)
{
    for(int k=0; k<n; k++)
        y[k] = x[k] + x[k+1]
}
```

(b) Array too small

```
int x[10];
c(x, 0);

#pragma css task output(x[20])
void c(int *x, int isLong)
{
    int len = isLong  20 : 10;
    for(int k=0; k<len; k++)
        x[k] = k;
}
```

(c) Output array too large

```
d(x, y);
// should wait here
// #pragma css wait on (x)
x[0] = 1;

#pragma css task input(x) output(y)
void d(int x[1], int y[1]) { ... }
```

(d) Missing wait

```
#pragma css task input(x,y) \
                 output(z)
void e(int x[10],
       int y[10],
       int z[10])
{
    for(int k=0; k<10; k++)
        x[k] = y[k] + z[k];
}
```

(e) Incorrect transfer direction

```
#pragma css task output(x[10])
void f(int *x)
{
    if (x == NULL)
        return; /* do nothing */
    /* ... */
}
```

(f) NULL pointer

Fig. 2. Example mistakes found by Starsscheck

In subfigure (b), the function reads outside array x. This example also shows how StarSs supports variable length arrays: by specifying the length in the pragma rather than the C prototype. The tool must therefore handle array sizes that are not known until run time. Conversely, in subfigure (c), if isLong==0, the array is declared larger than necessary, and neighbouring memory will be corrupted if the caller has allocated an array smaller than 20 elements.

Subfigure (d) is missing a pragma wait before the main program accesses array x. A wait is required even for write-after-read dependencies, because task creation returns immediately. The original contents of the array should remain unmodified until all tasks that read it have been allocated to SPEs, and outgoing DMAs have completed. All missing waits are race conditions, hence non-deterministic and notoriously hard to debug.

In subfigure (e), the direction of data transfer is incorrect. This bug will often be easy to find because it is so blatant, but it can also be found by Starsscheck. In subfigure (f), the author has not noticed that argument x can be NULL. While StarSs could be extended to allow NULL variable pointers, and pass them unmodified to the task, this is not the current behaviour. This example, and the example in subfigure (c) may cause exceptions deep inside the run-time library, for which source may not be available.

Starsscheck finds all the mistakes in Figure 2. It also has an option to check whether function arguments are correctly aligned for CellSs. This constraint is imposed by the Cell B.E. DMA engine [11, §7.2.1].

3 How Starsscheck Works

3.1 Overview

Figure 3 shows the structure of Starsscheck. A translation tool reads the StarSs annotations from the source code, generating a wrapper function for each task. It also translates the *finish* and *wait* pragmas into appropriate macros. Translation currently uses a Python script, but a more robust tool, based on Mercurium [12], is planned. The translated source code is compiled as normal, and executed under Valgrind, using the *Starssgrind* tool.

An alternative is to take an executable generated by the SMPSs compiler, and intercept all calls to the run-time library. The benefit would be that the program would not need recompilation, but only if you are using SMPSs, and not some other variant of StarSs, such as CellSs (since Valgrind does not support the Cell B.E.). This approach would work, but it would be specific to a certain version of the run-time API, which may change in future.

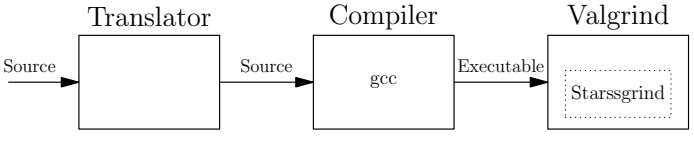


Fig. 3. Structure of Starsscheck

3.2 High-Level Interface to Starssgrind

Starsscheck uses Valgrind's client request mechanism, which recognises a "magic" sequence of instructions in the binary. The instructions normally do nothing, but are interpreted by Valgrind as a call into the analysis tool.

Table 1 lists the client request macros provided by Starssgrind. The precise semantics are given in Section 3.3, but the general idea is apparent in Fig. 4, which shows the translated version of the *bmod* function (Fig. 1).

Translation can be done using text substitution, without changing line numbers. Each task is given a wrapper function, which is a simple mechanism to define the region of stack it can touch, irrespective of the calling convention. The PUSH_CONTEXT macro enters a task. It sets up the task's context, which initially allows access only to the .text section and the stack below the wrapper function's frame pointer. The INPUT_BLOCK, OUTPUT_BLOCK and INOUT_BLOCK macros each declare a contiguous region of memory to be accessible by the task. The POP_CONTEXT macro leaves a task, and restores the master thread's context. The master thread is subsequently allowed read-only access to the task's input blocks, and no access to its output and inout blocks.

The WAIT_ALL and WAIT_ON macros support the finish and wait on pragma clauses, respectively. It is not necessary to translate the start pragma clause.

Table 1. Starsscheck client requests

Request	**Description**
PUSH_CONTEXT(void)	Enter task
INPUT_BLOCK(void *address, size_t len)	Declare input block
OUTPUT_BLOCK(void *address, size_t len)	Declare output block
INOUT_BLOCK(void *address, size_t len)	Declare inout block
POP_CONTEXT(void)	Return from task
WAIT_ON(void *address)	Restore given array
WAIT_ALL(void)	Restore all arrays

3.3 Starssgrind Contexts

Starssgrind, the Valgrind tool, maintains several *contexts*, which define the accessible regions of memory. The *current context* is currently active: all accesses to memory are checked against it, and bad accesses immediately generate a warning. The *all context* defines the whole memory space accessible by the sequential program. It is the initial context for the main thread, and it defines the memory that the main thread can pass to tasks. The *baseline context* is the starting point for tasks, which contains only the .text section. Inside a task, the *parent context* defines the context of the main thread. When an array is passed to a task, it is removed or rendered read only in the parent context. Such regions are moved to the *dead context*, which records the sizes of arrays, so that when the main thread performs a wait, the correct region of memory can be restored.

```
1  __attribute__((__noinline__))
2  void css_wrapped_bmod(float row[32][32],
3                        float col[32][32],
4                        float inner[32][32])
5  {
6    int i, j, k;
7    PUSH_CONTEXT();
8    INPUT_BLOCK(row, sizeof(float[32][32]));
9    INPUT_BLOCK(col, sizeof(float[32][32]));
10   OUTPUT_BLOCK(inner, sizeof(float[32][32]));
11   for (i=0; i<32; i++)
12     for (j=0; j<32; j++)
13       for (k=0; k<32; k++)
14         inner[i][j] -= row[i][k]*col[k][j];
15   POP_CONTEXT();
16 }
17
18 __attribute__((__noinline__))
19 void bmod(float row[32][32],
20           float col[32][32],
21           float inner[32][32])
22 {
23   css_wrapped_bmod(row, col, inner);
24 }
```

Fig. 4. Translated version of the *bmod* function from Fig. 1

A context is a disjoint set of *regions*, with each region covering a contiguous part of memory, with read-only or read-write access. The regions are stored in a balanced tree (our implementation uses a scapegoat tree [13]). Except within the dead context, adjacent regions with the same access rights get merged.

We chose to use a tree representation rather than shadow bits for three main reasons. The first reason is that we do not expect the compiler to generate accesses outside the supplied arrays. This is different from validity checking in *memcheck*, where copying an array containing uninitialized padding should make the destination padding uninitialized, rather than immediately generating warnings. The second reason is that, for realistic programs, there are few active regions, so it is feasible to use a tree; often even a list is sufficient. The third reason is that extending memcheck's efficient shadow bit representation to handle read-only regions and switching between contexts would be considerable work, with marginal benefit. See Section 4.2, however, for possible future work.

Figure 5 shows the current context before, during, and after task f. Assuming f is the first task created, the context before calling f is the same as the *all context*. During f, only the stack below its arguments, the .text section, and the argument a are visible. After calling f, a is not accessible until the main thread waits on a.

Since it is expensive to traverse the context tree for every memory access, we use a direct mapped region cache, based on the instruction address. The access is first checked against the region that the instruction previously hit. The region cache is cleared after any client request that switches the current context or deletes memory from it. The cache check is inserted directly into the VEX IR, which is the static single-assignment intermediate code used by Valgrind.

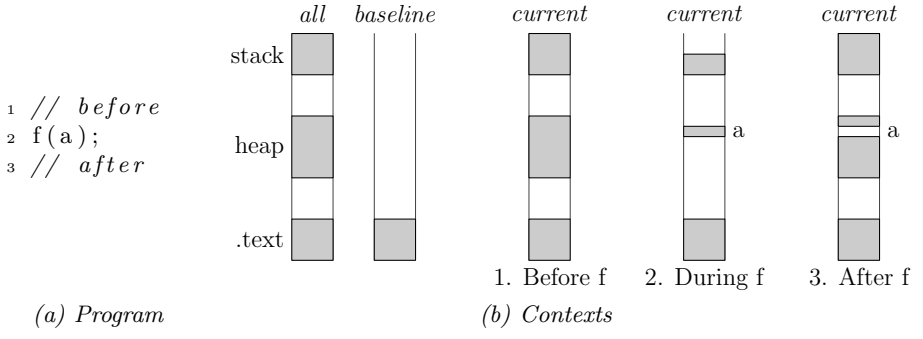


Fig. 5. Starssgrind Contexts

4 Evaluation

The most important consideration is the number of false positives and false negatives. Starsscheck finds all the mistakes in Fig. 2, as confirmed by our test cases, and does not create any false positives, except as discussed in Section 4.3.

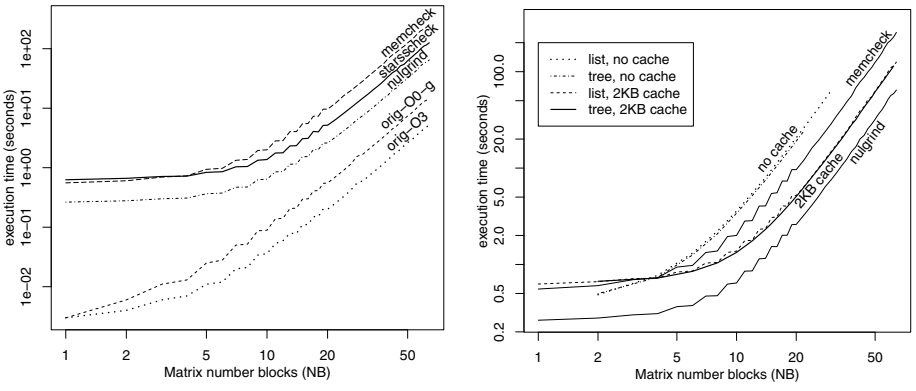
4.1 Performance

A tool such as Starsscheck will only be used if it is reasonably fast. Our experiments show that unless tasks are so small that StarSs itself is slow, the execution time under Starsscheck is similar to memcheck, which should be acceptable.

Fig. 6 shows the execution times for the sparse LU factorization example from the StarSs distribution, using the default block size of 32×32. Subfigure (a) shows the execution time for square matrices up to 64×64 blocks. For matrices of size about 640×640 (20×20 blocks) or above, the slowdown of Starsscheck is close to 8 times compared with the original, unoptimized code. For comparison, memcheck is about 16 times slower than the original, and *nulgrind*, which is Valgrind without instrumentation, is about four times slower.

Subfigure (b) compares four variants of Starsscheck, with the region cache enabled and disabled, and using either a tree or a linked list to hold the set of regions. The difference between the tree and linked list is insignificant for this example, but the cache gives a speed-up of about 3.5, if the matrix size is 16 blocks or more. The mean number of regions is approximately 9, requiring an average search depth of 1.5. The average number of regions decreases slightly as the matrix increases, because a greater proportion of time is spent in tasks, which have a below average number of valid regions.

Fig. 7(a) shows the "nasty" benchmark, which is intended to show worst case performance. This benchmark has extremely fine grain tasks: each task contains a single arithmetic statement. Starsscheck has high overhead because StarSs itself has high overhead. The execution times are shown in Subfigures (c) and (d), and Starsscheck is much slower than memcheck. The region cache provides little benefit, because every access to the `a` array misses. The average search



(a) Starsscheck in context *(b) Starsscheck variants*

Fig. 6. Performance results for Sparse LU factorization

depth in the tree increases logarithmically in the number of tasks, which is the worst case.

Unrolling is a standard technique to increase task granularity. The two loops in Fig. 7(b) have been unrolled by a factor of 1,024, which requires a rearrangement of the a array. Subfigure (e) shows the execution time of the modified benchmark, which is comparable to memcheck.

4.2 Limitations

The main limitation of Starsscheck is that, unlike memcheck, it does not track the validity of data. Fig. 8(a) shows a task with an output block that should be marked *inout*. There is in fact, inside Starsscheck, no difference between OUTPUT_BLOCK and INOUT_BLOCK, since both allow the task to read and write, and both make the block inaccessible to the main thread until it waits.

Fortunately, this error can be found using memcheck. We recommend running under memcheck in any case, because memcheck finds errors that are outside the scope of Starsscheck. The Starsscheck distribution will include a translator that invalidates memory corresponding to each output array. It does so using the VALGRIND_MAKE_MEM_UNDEFINED memcheck client request. Clearly the functionality of memcheck and Starsscheck could be combined into a single tool.

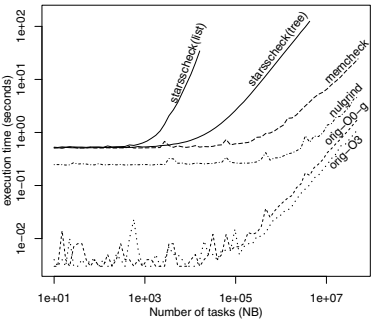
The current implementation of Starsscheck fully supports CellSs, and a subset of SMPSs. SMPSs introduces *array region specifiers*, which describe a region in a multi-dimensional array, and *opaque pointers*, which are ignored by the runtime, and allow tasks to exploit the shared memory hardware. Array region specifiers require a straightforward extension to bounds checking. Starsscheck should skip validity checking for addresses calculated via opaque pointers. This requires tracking, either using static analysis or *shadow bits* similar to memcheck. Tracking of opaque pointers is orthogonal to the rest of the tool.

```
1  #pragma css task inout(p)
2  void f(int *p)
3  {
4    p[0] += 1;
5  }
6
7  int *nasty(void)
8  {
9    int *a = malloc(sizeof(int[NB*2]));
10   int j;
11   memset(a, 0, sizeof(int[NB*2]));
12
13 #pragma css start
14   /* Process even elements */
15   for(j=0; j < NB*2; j += 2)
16     f(&a[j]);
17
18   /* Process odd elements */
19   for(j=1; j < NB*2; j += 2)
20     a[j] += 1;
21 #pragma css finish
22   return a;
23 }
```

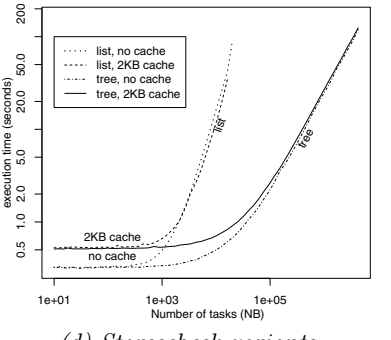
(a) Source code (nasty)

```
1  #pragma css task inout(p)
2  void f(int p[1024])
3  {
4    int k;
5    for(k=0; k<1024; k++)
6      p[k] += 1;
7  }
8
9  int *nasty1k(void)
10 {
11   int *a = malloc(sizeof(int[NB*2048]));
12   int j,k;
13   memset(a, 0, sizeof(int[NB*2048]));
14
15 #pragma css start
16   /* Process even blocks */
17   for(j=0; j < NB*2048; j += 2048)
18     f(&a[j]);
19
20   /* Process odd blocks */
21   for(j=1024; j < NB*2048; j += 2048)
22     for(k=0;k<1024;k++)
23       a[j+k] += 1;
24 #pragma css finish
25   return a;
26 }
```

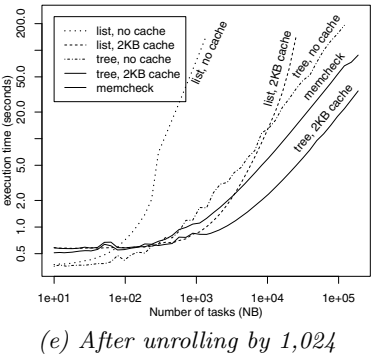
(b) Source code (nasty1k)



(c) Starsscheck in context



(d) Starsscheck variants



(e) After unrolling by 1,024

Fig. 7. Worst case "nasty" benchmark

```
#pragma css task output(x)
void b(int x[1])
{
    x[0] += 1;
}
```

(a) Argument should be marked inout

```
#pragma css task input(x)
void f(int x)
{
    printf("x is %d\n", x);
}
```

(c) printf does not respect StarSs

```
#pragma css task input(s) output(y)
void f(char s[13], int y[1])
{
    y[0] = strlen(s);
}
```

(b) SIMD strlen reads outside array

Fig. 8. Potential false negatives and false positives

4.3 Eliminating False Positives

Fig. 8(b) shows an example where Starsscheck can generate spurious warnings. A SIMD implementation of strlen may read memory above and below the string itself (but inside the same memory page). Note that memcheck also has difficulty following some C library functions [7], and we can use the existing Valgrind machinery to replace functions such as strlen.

Also, certain C library functions, such as printf, do not respect the StarSs memory model. We use the *suppressions* mechanism of Valgrind to suppress any warnings that arise.

5 Related Work

We are unaware of any other tools that check the input and output definitions of task-based programs. There are, however, several tools that find data races in shared memory. The crucial difference is that Starsscheck verifies information needed by the StarSs run-time for the program to work at all, whereas the data race detectors discover unordered, and therefore racing, accesses to the same location in memory. Of the motivating examples in Fig. 2, only subfigure (d) is a data race. The other errors cause the run-time to fail to transfer the correct data in or out of the task.

The Cilk **Nondeterminator** [14] finds data races in Cilk programs. A nondeterministic Cilk program is not necessary wrong, since the language provides locks and it allows communication through shared memory [15]. Nondeterminator checks that programs that are supposed to be deterministic are in fact so. The algorithm assigns an ID to each task at runtime, and maintains, for every location in memory, the IDs of its most recent writer and some previous reader. It checks whether accesses are ordered via Cilk's shared memory model. Its "SP-bags" algorithm assumes that dependencies between tasks follow a series-parallel

DAG, which is true for Cilk but not for StarSs. Hence it is unlikely to be possible to adapt the algorithm for our purposes.

Many tools find data races in multi-threaded programs. **Helgrind** [6] uses the Eraser algorithm [16] to find data races in POSIX pthreads programs. **ThreadSanitizer** [17] uses a hybrid algorithm based on happens-before and locksets. **CORD** [18] and **ReEnact** [19] are hardware techniques to detect data races. **MTRAT** [20] is a race detector for Java. Since a StarSs program is ultimately implemented using threads, these tools could find the data race in Fig. 2(d), but they could not check whether the StarSs pragma annotations are correct.

6 Conclusions

This paper presents Starsscheck, a tool based on Valgrind, which finds undefined behaviour in a task-based parallel program.

Starsscheck currently supports Star Superscalar, an extension to C that uses pragma annotations to mark functions that can be executed in parallel. If the annotations are wrong, the program may exhibit race conditions or exceptions inside the run-time system. Starsscheck uses binary translation to discover program behaviour that is inconsistent with the pragmas. It can be adapted for similar programming models such as TPC.

Starsscheck finds many common errors, as described in the paper. Except in pathological cases where StarSs itself is slow, execution time is comparable with memcheck, Valgrind's default tool.

Acknowledgements. This work has been supported by the Spanish Ministry of Science and Innovation under the Consolider contract number TIN2007-60625, the European Network of Excellence HIPEAC-2 (ICT-FP7-217068), the ENCORE project (ICT-FP7-248647) and the IBM-BSC MareIncognito project.

References

1. Bellens, P., Perez, J., Badia, R., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. ACM, New York (2006)
2. Pham, D., Behnen, E., Bolliger, M., Hofstee, H., et al.: The design methodology and implementation of a first-generation Cell processor: a multi-core SoC. In: Custom Integrated Circuits Conference, pp. 45–49 (2005)
3. Perez, J., Badia, R., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: 2008 IEEE International Conference on Cluster Computing, pp. 142–151 (2008)
4. Barcelona Supercomputing Center: Cell Superscalar (CellSs) User's Manual Version 2.2 (2009)
5. Barcelona Supercomputing Center: SMP Superscalar (SMPSs) User's Manual Version 2.0 (2008)
6. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI, pp. 89–100 (2007)

7. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, USENIX Association, p. 2 (2005)
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005, pp. 190–200 (2005)
9. Tzenakist, G., Kapelonis, K., Alvanost, M., Koukost, K., Nikolopoulost, D., Bilast, A.: Tagged Procedure Calls (TPC): Efficient runtime support for task-based parallelism on the Cell Processor. In: HiPEAC 2010. LNCS, vol. 5952. Springer, Heidelberg (2010)
10. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. ACM SigPlan Notices 30(8), 207–216 (1995)
11. IBM: Cell Broadband Engine Programming Handbook including PowerXCell 8i Version 1.11 (2009)
12. Balart, J., Duran, A., Gonzalez, M., Martorell, X., Ayguade, E., Labarta, J.: Nanos Mercurium: a Research Compiler for OpenMP. In: Proceedings of the European Workshop on OpenMP, vol. 2004 (2004)
13. Galperin, I., Rivest, R.: Scapegoat trees. In: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 165–174. Society for Industrial and Applied Mathematics, Philadelphia (1993)
14. Feng, M., Leiserson, C.: Efficient detection of determinacy races in Cilk programs. Theory of Computing Systems 32(3), 301–326 (1999)
15. MIT LCS: Cilk 5.4.6 Reference Manual (1998)
16. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (TOCS) 15(4), 391–411 (1997)
17. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer—data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, pp. 62–71 (2009)
18. Prvulovic, M.: Cord: cost-effective (and nearly overhead-free) order-recording and data race detection. In: The Twelfth International Symposium on High-Performance Computer Architecture, pp. 232–243 (February 2006)
19. Prvulovic, M., Torrellas, J.: ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In: Annual International Symposium on Computer Architecture, vol. 30, pp. 110–121 (2003)
20. IBM: Multi-Thread Run-time Analysis Tool for Java, `http://www.alphaworks.ibm.com/tech/mtrat`