Pure Functions in C: A Small Keyword for Automatic Parallelization

Tim Süß, Lars Nagel, Marc-André Vef, André Brinkmann
Zentrum für Datenverarbeitung
Johannes Gutenberg University Mainz
Mainz, Germany
{t.suess, nagell, vef, brinkman}@uni-mainz.de

Dustin Feld, Thomas Soddemann
Fraunhofer SCAI
Schloss Birlinghoven
Sankt Augustin, Germany
{dustin.feld, thomas.soddemann}@scai.fraunhofer.de

***Abstract*—The need for parallel task execution has been steadily growing in recent years since manufacturers mainly improve processor performance by scaling the number of installed cores instead of the frequency of processors. To make use of this potential, an essential technique to increase the parallelism of a program is to parallelize loops. However, a main restriction of available tools for *automatic* loop parallelization is that the loops often have to be 'polyhedral' and that it is, e. g., not allowed to call functions from within the loops.**

In this paper, we present a seemingly simple extension to the C programming language which marks functions without side-effects. These functions can then basically be ignored when checking the parallelization opportunities for polyhedral loops. We extended the GCC compiler toolchain accordingly and evaluated several real-world applications showing that our extension helps to identify additional parallelization chances and, thus, to significantly enhance the performance of applications.

I. Introduction

Processor vendors can no longer improve performance cost-effectively by simply scaling processor frequencies. Instead, they have to increase the number of cores per processor. Furthermore, vector units like SSE or AVX are integrated to perform multiple arithmetic operations in parallel. Both architectural trends are evidence for the rising importance of parallel processing over the last years.

To make use of these capabilities, programs have to run 'as parallel as possible'. One approach is to split programs into many parallel threads and distribute them among the processor cores. For this, programmers can use libraries and application programming interfaces, such as *OpenMP*, *OpenACC*, or *Cilk++* [1], [8], [19]. Unfortunately, threads and vector units are typically insufficiently used as programmers need a deep understanding of these libraries and parallelism in general.

Several research projects are focused on *automatic parallelization tools* that transparently transform sequential source code into parallel code. Existing tools, such as *Par4all*, *PIPS*, and *PluTo*, are able to parallelize sequential program parts at certain conditions [3], [6], [25], [27]. For instance, PluTo is capable of transforming a nested loop if it is *polyhedral*, i.e., *all* array accesses within the loop are affine functions of the loop iterators (for details see Section III-0c). Under these circumstances, it is possible to create the *polyhedral model* of the nested loop and perform loop nest transformations and optimizations. Library-specific pragmas for parallelization and vectorization are automatically inserted and, in some cases, even memory accesses can be optimized to achieve a better cache usage [10], [11].

Although there has been significant progress on automatic parallelization, the tools still induce a number of restrictions: 1) Sections to be parallelized are generally not allowed to contain functions calls. 2) Current transformers can only transform polyhedral codes which requires complete knowledge of memory accesses at compilation time. 3) Many transformers require parallel section candidates to be marked manually by the programmer (however, this is not always necessary [4]).

Besides using libraries and parallelization tools, there are several other approaches to extend programming languages for automatic parallelization. Functional languages like Haskell, for example, inherently allow the parallel execution of functions [20], [22]. They benefit from their paradigm's property that (most) functions have no side-effects. But at the same time, they suffer from their lower performance compared to languages normally used in high performance computing like *Fortran* and *C* [13].

On the other hand, functions in *Fortran* and *C* can have side-effects which makes it difficult or impossible to parallelize them (automatically). For this reason, Fortran introduced the `pure` keyword. Pure functions are guaranteed to be free of side-effects and allow Fortran compilers to parallelize more code segments automatically. Prior to this work, such a feature was not available for C. One reason is that testing if a function is free of side-effects is more difficult than in Fortran.

This work adds the `pure` keyword to the C language and shows how programs and source-to-source transformers can benefit from it. The new keyword is used similarly to other existing function prefixes or modifiers such as `static` or `inline`. For this we developed an additional compiler pass which verifies that functions marked as `pure` do not change the state of any variable outside their scope. Thus, it ensures that these functions have no undesired *side-effects*. Using the *PluTo framework* [6], we demonstrate the power of this new feature and test our approach with real-world applications.

Like other polyhedral code transformers, our solution requires slight code modifications, but using the new keyword has additional benefits. The compiler's optimizer can, e.g., exploit that parameters and their content will never be modified.

Moreover, the `pure` keyword can also mark library functions as side-effect-free with the effect that even library function calls can be used in automatically parallelized program parts.

II. Related Work

In this section, we discuss C and Fortran language extensions and tools that support automatic parallelization.

A. Fortran and C Language Extensions

There are different extensions and features of programming languages enabling or supporting automatic parallelization of program parts. Fortran provides two concepts: the *pure* keyword [7] and *co-arrays*, which became part of the Fortran 2008 standard [21], [24]. C provides the function attributes `__attribute__((const))` and `__attribute__((pure))` [2] which, however, are mere programmer hints to the compiler. In comparison to our compiler chain, there is no further semantic analysis to verify the 'const-ness' of a function so that side-effects are possible.

High Performance Fortran (HPF) is a language extension which is not part of the Fortran standard. HPF reads directives from the code and tells the compiler how data have to be distributed and processed [17].

B. Parallelization Tools

Our compiler chain includes tools that automatically transform, optimize and parallelize C source code based on the polyhedral model. This model has been well studied and numerous source-to-source compilation tools have evolved, such as *PluTo* [5], *PPCG* [27], *Par4ALL* [25], or the *ROSE compiler infrastructure* [23] with its *PolyOpt/C* optimizer. These frameworks traditionally aim for an automatic OpenMP and SIMD parallelization of sequential CPU codes; some (e.g., PPCG) are also capable of generating CUDA or OpenCL code for GPUs. Our code generator is a consolidation of PluTo and its *SICA* extension (available from the 'SICA' branch of the PluTo Git repository). SICA extends the PluTo framework in terms of highly optimized SIMD and multi-core code generation (here denoted as *PluTo-SICA*) [10], [11].

III. The Integration of Pure Functions into C

When functions in a computer program are executed in parallel, they may interfere with each other, e.g., by altering shared class variables. However, if functions do not have such side-effects, it is generally fail-safe to run them in parallel. Nevertheless, most polyhedral source-to-source transformers are not able to parallelize loops that contain side-effect-free function calls as they are not encapsulated in a larger framework, which holds additional information about '*the outside*' of their concrete scope. This is due to memory accesses that must be known during the transformation process, but function calls mask this information. By introducing the `pure` keyword, we allow the programmer to mark side-effect-free functions and the compiler to verify that these functions are really *pure*, i.e., free of side-effects. The transformer can consider this and ignore dependencies within these functions to, finally, potentially parallelize surrounding loops.

Listing 1. Valid and invalid operations in `pure` functions.

```
int* globalPtr;
void func1();
pure int* func2(pure int* p1, int p2) {
  int a = p2;
  int* c = (int*) malloc(3*sizeof(int));
  pure int* ptr = p1;
  int* extPtr1 = globalPtr;           // invalid
  pure int* extPtr2 = (pure int*)globalPtr;
  func1();                            // invalid
  pure int* extPtr3 = (pure int*)func2(p1,p2);
  return c;
}
```

a) Language Extension: Many functional languages allow automatic parallelization by exploiting the properties inherent in this programming paradigm. In such languages, a function can be seen as a black box: It is supplied with parameters as input and returns a result. By construction, there are no side-effects. This usually does not hold for imperative programming languages like C. A C function is not a function in the mathematical sense because it can affect program parts outside of the function's scope.

We extended C with an additional function modifier: `pure`. Functions of this type mimic the behavior of functions in functional programming languages as they have no impact on the program's state except for the results of the performed computation. All elements of the surrounding program have the same state after the execution as they had prior to the function call.

The `pure` keyword is placed in front of a function to label it as pure as well as in front of pointers, like that:

```
pure int* func(pure int* p1, int p2);
```

It is important to note, that `pure` pointers cannot be modified, nor can their content. They can only be assigned once. Therefore, it is generally not necessary to allocate their memory dynamically with the `malloc` function to make it accessible to the outside.

Listing 1 shows allowed and denied operations in `pure` functions. `pure` functions can only call other `pure` functions. Global pointers (e.g., `globalPtr`) can be used after being type-casted and assigned to a local pointer (`extPtr2`).

b) Compiler Pass: The implementation of our additional compiler pass is almost exclusively based on standard tools, e.g., the *GCC* tool chain and the *AntLR 4.5* compiler (or parser generator). The *AntLR* repository provides a C grammar built from the *C11* specification. We can, therefore, assume that the C programming language standard is not harmed.

Our compiler pass must perform a syntactical and semantical analysis. It receives a C file that has been preprocessed by our own and the GCC preprocessor. Hence, all required headers are included, all `defines` are substituted, etc. This file is submitted to our preprocessor to generate an abstract syntax tree (AST). The AST is traversed while most of the code is ignored, unless a function declaration or implementation is marked as `pure` or a *for*-loop is traversed.

If a function is declared or implemented `pure`, the function name is added to a set containing all these functions. The set is required to check if a `pure` function only calls other `pure`

functions or appropriately defined side-effect-free functions. We initialize the set with the C standard functions that have no side-effects (e.g., `sin`, `cos`, `log`, etc.). Additionally, we have to insert `malloc` and `free` to the set although these functions are not strictly free of side-effects to return more complex structures.

The compiler pass also verifies that assignments do not modify function-external data: If a pointer assigns function-external data (e.g., in form of parameters or global data), it must be declared `pure`, and the assigned data requires a respective type cast with the prefix `pure`.

If data is stored somewhere in the function, our compiler pass checks for storage initialization in the function's scope as well. If the target was declared outside of the scope, it would initiate a side-effect and therefore trigger a compilation error.

If our compiler pass finishes without errors, it is ensured that the `pure` functions do not have any undesired side effects. However, the `pure` keyword would cause a compiler error in the classical GCC tool chain and must be removed or exchanged before proceeding. The prefixes of the pointers in a function's parameter list can be replaced by `const`. Additionally, the function prefix must be removed.

An important property of our extension is that it does not negatively influence the C programming language. Removing it does not affect the results of a program other than that it might not be as parallelizable as before.

c) Automatic Parallelization: During the compiler pass, each internally called function in a `for`-loop is analyzed and checked if it only calls `pure` functions. If this is the case, we surround the loop by the keywords `#pragma scop` and `#pragma endscop`. Such loops are not allowed to contain function calls in the following stages of the compiler chain to ensure that they can be processed by the parallelization tools. Hence, we substitute function calls in such loops by special, unique words to make the function calls appear as if they were constants. This way, the marked sections can be checked (in our case, by PluTo in polycc) whether they are polyhedral. If such a section is polyhedral, the transformer inserts pragmas for OpenMP and for vectorization.

A valid (*legal*) transformation of nested loops results in a new execution order of the iteration points respecting the data-dependencies in the underlying polyhedral model [9] [14] [15]. In other words, the original ordering is preserved if an iteration point computes results that are needed as an input to another iteration point. The transformation may include the manipulation of loop dimensions (index variables) leading to a deformation of the polyhedron such that computations can be processed in parallel.

After the polyhedral transformer has finished its tests and (if necessary) its transformations, the previously substituted function calls are adapted and reinserted into the source code. Since PluTo inserts new program parts, including preprocessor directives, we start the GCC toolchain from the beginning with the program file built at the end of our compiler pass.

Automatic code parallelization tools require that accesses made within a nested loop are affine and statically comprehensible. Listing 2 shows one counterexample as the function call is not statically analyzable by PluTo. This issue of a relatively limited view of many source-to-source tools can partially be solved by using `pure` annotations together with our compiler pass. For each parameter of a `pure` function, the compiler pass checks whether it also appears on the left-hand side of an assignment operator in the loop nest and, therefore, recognizes these *write dependencies*. If this is the case, the code cannot directly automatically parallelized.

Yet, the compiler pass can be tricked by aliases. Similar to other performance-relevant optimizations suffering from aliasing, this construct disrupts our approach as well. Aliasing obscures using the same memory region under different names. Although there are static code analyzers for detecting such pointers at compilation time, there are situations where these tools fail, e.g., if the alias depends on runtime conditions [16] [28]. Other tools instrument program codes and detect (*at runtime*) if an array is accessed by using different pointers [12].

IV. Evaluation

We evaluated our approach by applying it to two different program codes. The first application multiplies two matrices with 4096×4096 elements each. The `pure` version uses a nested function calculating a dot product. This version cannot be parallelized by standard polyhedral transformers. As a result, the dot product is manually inlined in the competing PLuTo / PluTo-SICA version to allow any comparison at all. Additionally, we ran the matrix-matrix multiplication with a hand-tuned version of the *Intel Math Kernel Library* (MKL).

The second application is a standalone version of the *ELL sparse matrix vector multiplication* function extracted from the LAMA library [18]. We used the *Boeing/pwtk*[1] data set as input which contains a symmetric matrix consisting of over 217K rows and columns with 11.5 million non-zero elements (about 155 MiB). The loop nest of the LAMA function contains a function call and indirect addressing (because of the sparseness of the matrices). Finally, the inner `pure` function computes the dot product of two vectors.

A. Test Environment

The tests were performed on a computer equipped with four AMD Opteron 6272 processors (16 cores @2.1 GHz each) and 512 GiB of RAM. The application were compiled with *GCC 4.4.7 20120313* (*Red Hat 4.4.7-3*) and *Intel C/C++ Compiler*

[1] http://www.cise.ufl.edu/research/sparse/MM/Boeing/pwtk.tar.gz

Listing 2. Invalid use of a `pure` function. The function call within the loop nest violates the properties of polyhedral loops.

```
pure int func(pure int* _a, int _idx) {
   return _a[_idx-1]+_a[_idx];
}
void main() {
   int arr[100];
   ...
   for (int i = 1; i < 100; i++) {
      arr[i] = func(arr, i); // invalid assignment
   }
}
```

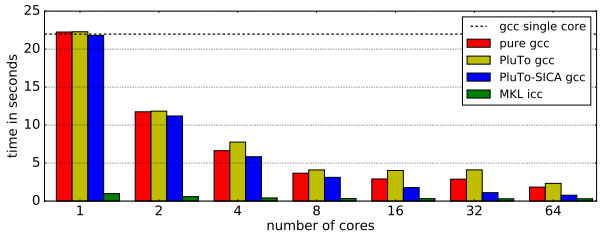

Fig. 1. Execution time of the matrix-matrix multiplication using GCC.

(ICC) 16.0.2 and used the `-O2` flag in all cases to retain accuracy and comparability to the `O2`-compiled MKL.

B. Scaling Tests

We evaluated the scaling behavior of the applications. For each program we exponentially increased the number of parallel cores from 2^0 to 2^6 and measured the average runtime over 20 application runs.

a) Matrix-matrix multiplication: Using the GCC-based compiler chain for the matrix-matrix multiplication, the first test compares the different parallelization tools with each other and with the corresponding MKL version compiled with the Intel compiler (see Fig. 1). The sequential GCC-compiled version took 21.97 seconds (dashed line).

With an increasing number of utilized cores, the execution time is strictly decreasing when our compiler chain is used (pure bars). It temporarily increases scaling from 16 to 32 cores when the PluTo polyhedral transformer is used on its own (PluTo bars). Furthermore, the pure version is always significantly faster the the 'simple' PluTo parallelization.

At first glance, the results may seem counter-intuitive as `pure` uses PluTo to parallelize the code in its tool chain. Furthermore, PluTo inlines the function, which is typically considered to be faster than calling the function [26]. We therefore analysed the parallelized source code and found out that another program part was parallelized using the `pure`-directive although it was not planned to run concurrently: As mentioned, the `malloc` operation is one of the C standard functions that we mark `pure`, and a loop allocating the matrices with `malloc` was therefore parallelized, too.

Nevertheless, our automatic parallelization using the GCC compiler chain cannot compete with PluTo-SICA and is much slower than the hand-tuned MKL. This is because PluTo-SICA's code and the MKL implementation can make exhaustive use of SSE/AVX directives and cache-align the data better.

The `pure` version experiences a significant performance advantage when ICC is used while the PluTo version does not benefit from it (see Fig. 2) on only a few cores for our tested matrix size. The performance improvement is higher for smaller core numbers, while the performance of `pure` together with the ICC compiler converges to the performance of the GCC compiler chain for core counts higher than 16 cores. The performance of these two parallelization approaches is only faster than the ICC single-core version when more than two cores are used. PluTo-SICA is only able to (clearly) outperform the `pure` directive for eight or more cores.

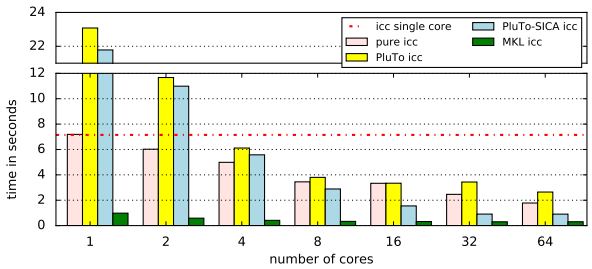

Fig. 2. Execution time of the matrix-matrix multiplication using ICC.

The comparison with the matrix-matrix multiplication using MKL shows that the hand-tuned code can still significantly outperform all other program versions and that there is still optimization potential for automatic parallelization tools. The MKL version is 7.28 times faster than the `pure` version for a single core and 5.82 faster in the case of 64 cores.

b) LAMA (ELLMatrix): For the LAMA application, we can only provide results for manually modified code and for code automatically generated with our `pure` compiler chain. The PluTo and the PluTo-SICA tools are unable to parallelize this code without our compiler pass.

The runtimes for the ELL sparse matrix vector multiplication with an increasing number of cores are shown in Fig. 3. While the versions compiled with Intel's ICC are more efficient than the corresponding GCC versions for less than 16 cores, they are less efficient for more than 16 cores. Generally, our automatic parallelization achieves a performance comparable to the one of the manually built executable.

V. Conclusion

We have introduced the `pure` keyword for the C programming language which marks side-effect-free functions and helps to automatically parallelize C programs. Prior to our implementation, polyhedral transformers were generally unable to parallelize any loops that contain function calls. With our extension, it is now fail-safe for polyhedral transformers to ignore `pure` functions so that it becomes possible to parallelize more classes of loop-nests. In our evalution we have shown that our preprocessor automatically accelerates programs if their main computation is embedded in a polyhedral loop.

Although there exist other language extensions, this is the first working C compiler pass which guarantees that a function has no side-effects. In the future we will integrate the `pure` keyword into the *C++* programming language and also provide a tighter coupling between `pure` and PluTo-SICA to provide better cache alignment and better support for code vectorization.

Acknowledgments

This work was supported by the German Ministry for Education and Research under grant 01|H13004A (FAST).

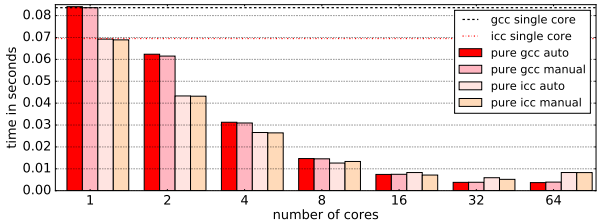

Fig. 3. Execution time of the LAMA application.

REFERENCES

[1] *The OpenACC Application Program Interface. Version 2.5*. 2012. http://www.openacc-standard.org/.

[2] ARM Compiler toolchain Compiler Reference - Compiler-specific Features, 2015. *http:// infocenter.arm.com/help/topic/com.arm.doc.dui0491c/CHDFIJAE.html*.

[3] M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, and P. Villalon. PIPS Is not (just) Polyhedral Software Adding GPU Code Generation in PIPS. In *First Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO 2011*, Chamonix, France, Apr. 2011.

[4] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. *Putting Polyhedral Loop Transformations to Work*, pages 209–225. 2004.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pages 101–113, 2008.

[6] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, Oct. 2007.

[7] I. Corporation. *Intel Fortran Language Reference*. 2001. Document Number: 253261-002.

[8] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program. volume 21*, pages 313–348, 1992.

[10] D. Feld, T. Soddemann, M. Jünger, and S. Mallach. Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation. In A. Größliger and L.-N. Pouchet, editors, *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, pages 45–54, Berlin, Germany, Jan 2013.

[11] D. Feld, T. Soddemann, M. Jünger, and S. Mallach. Hardware-aware automatic code-transformation to support compilers in exploiting the multi-level parallel potential of modern cpus. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores, COSMIC '15, San Francisco Bay Area, CA, USA, February 8, 2015*, pages 2:1–2:10, 2015.

[12] R. Gad, T. Süß, and A. Brinkmann. Compiler Driven Automatic Kernel Context Migration for Heterogeneous Computing. In *34th International Conference on Distributed Computing Systems (ICDCS)*, 2014.

[13] A. Gill and C. Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 1–12, New York, NY, USA, 2007. ACM.

[14] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program. volume 34*, pages 261–317, 2006.

[15] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004.

[16] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 254–263, 2001.

[17] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of high performance fortran. *Commun. ACM*, 54(11):74–82, Nov. 2011.

[18] J. Kraus, M. Förster, T. Brandes, and T. Soddemann. Using lama for efficient amg on hybrid clusters. *Computer Science - R&D*, 28(2-3):211–220, 2013.

[19] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[20] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. *SIGPLAN Not.*, 44(9):65–78, Aug. 2009.

[21] R. W. Numrich and J. Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, Aug. 2005.

[22] L. Petersen, D. Orchard, and N. Glew. Automatic simd vectorization for haskell. *SIGPLAN Not.*, 48(9):25–36, Sept. 2013.

[23] M. Schordan and D. J. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC*, pages 214–223, 2003.

[24] A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty. Fortran 2008 coarrays. *SIGPLAN Fortran Forum*, 34(1):10–30, Apr. 2015.

[25] SILKAN. HPC project. Par4All automatic parallelization. *http://www.par4all.org*.

[26] R. M. Stallman and G. DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.4.7*. CreateSpace, Paramount, CA, 2009.

[27] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4), 2013. Selected for presentation at the HiPEAC 2013 Conf.

[28] Z. Xu, T. Kremenek, and J. Zhang. *Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, chapter A Memory Model for Static Analysis of C Programs, pages 535–548. 2010.