RZ 3834 (# Z1211-001) 11/13/2012
Computer Science 8 pages

Research Report

A Tool for Analysis and Visualization of Application Properties

Victoria Caparrós Cabezas

IBM Research – Zurich
8803 Rüschlikon
Switzerland

Research
Almaden • Austin • Brazil • Cambridge • China • Haifa • India • Tokyo • Watson • Zurich

A Tool for Analysis and Visualization of Application Properties

Victoria Caparrós Cabezas
IBM Research—Zürich
CH-8803 Rüschlikon, Switzerland

***Abstract*—Performance modeling comprises two important areas. First, it is necessary to identify and theoretically analyze *what* are the properties of the applications that impact performance, and *how* they impact performance. This requires the definition of mathematical or probabilistic models of the applications properties, or the application of advanced techniques, such as machine learning, to model and predict their impact in performance. The other aspect of performance analysis is to define the mechanisms to actually *measure* application properties. These approaches range from analysis of algorithms complexity to evaluation of the application execution on real hardware.**

This report addresses the last topic and presents a tool developed within IBM for quantification of application properties. It integrates, in a single tool, several analyses of properties that were previously done independently—parallelism and data reuse—, and it extends traditional single-value metrics and reports properties over the entire execution. It is based on the LLVM compiler infrastructure as it enables an analysis that is language- and microarchitecture-independent.

This document serves as the deliverable for the documentation of characterization tools (item 3.15 in version 4 of the planning Gantt charts), and the tool described herein is part of the deliverables in Milestone 5 in version 4 of the planning Gantt charts.

I. Introduction

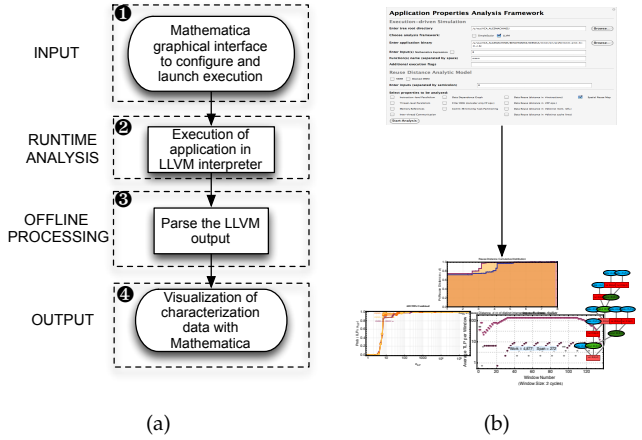
Techniques for measuring application's properties for performance evaluation and prediction range from theoretical analysis of computation and communication of algorithms [8, 19], to detailed microarchitectural simulation [2], or collection of empirical data on the application running on a specific platform using hardware performance counters [9].

Our characterization tool for understanding application's inherent properties is based on previous studies that use a microarchitectural simulator to emulate a machine with unlimited hardware resources, and quantify application behavior from the analysis of the data dependences and data movement properties of the dynamic instruction trace during execution on the simulator [6, 17]. This approach has several advantages with respect to the aforementioned techniques. First, application properties are measured for the particular input considered, as opposed to theoretical analysis of the algorithm, which do not consider input size or model it statistically. Second, it provides a better insight into application behavior, since it exposes a broader range of application properties, not only those that are exploitable with existing microarchitectural features (what would have been measured with hardware performance counters), but also properties that may require new hardware architectures in order to be exploited. Finally, this approach enables us to reason about application's performance across different platforms with just a single pass of the analysis, not requiring to repeat the analysis for every hardware configuration of interest. A precursor of the characterization framework presented in this work was actually built using SimpleScalar [4], a microarchitectural simulator that has extensively been used for microprocessor design evaluation. It was a useful previous step because SimpleScalar had been used in previous limit-case parallelism studies [12], and we could verify the output of our characterization against previous published results.

Whereas these data-flow and data reuse analyses are microarchitecture-independent, still are performed on a particular ISA (MIPS in the cited references), which introduces some dependences in the analyzed binary. For example, the limited number of registers in the MIPS architecture will generate some register spills—and, hence, some extra memory accesses—that, in a different machine with enough number of registers, would have not been measured, impacting the data movement properties of the application. The new characterization tool attempts to overcome this ISA dependence by doing the analysis on an interpreter that executes a dynamic trace of instructions expressed in an *Intermediate Representation* (IR). This IR is an intermediate stage in the translation process from source code to machine code, and can be viewed as a platform-independent assembly language. We will take advantage of this language- and machine-agnostic representation to quantify application properties, and keep the analysis free from specific features of the programming language in which applications have been implemented (so far, the tool works with C and C++ sources), and from low-level architectural details such as memory addressing modes or calling conventions.

The characterization tool consists of the following components:

- A graphical interface implemented in Mathematica



(a) (b)

Fig. 1. (a) Stages of the application analysis flow . (b) Graphical interface that simplifies the tool usage.

to configure and launch the execution of the application in the interpreter.

- A set of C++ files that are integrated into the LLVM compiler infrastructure and implement the application analysis at the LLVM IR level when the application is executed in the interpreter.
- A set of routines included in a Mathematica package that process the LLVM output and generate the graphics.
- Visualization of the characterization output in Mathematica.

Figure 1(a) depicts these components and shows the general flow of the tool. Note that stages one, three, and four can be omitted and the analysis can be done directly with the LLVM interpreter command line. However, the framework usage can be simplified by using the graphical interface (Figure 1(b)).

A. Contributions

The contributions of this work are the following:

- A characterization **framework** that quantifies two properties that traditionally have been studied independently—**parallelism and data reuse**. Although within the framework they are analyzed separately, i.e., their interaction is not studied, it is important to simultaneously have these two properties, which we consider may be the main limiters to performance and power scaling on future computing platforms.
- The characterization is done at a **language- and machine-agnostic** level of abstraction of the application, what enables us to reason about **inherent properties**, i.e., properties that are independent from the hardware.
- Finally, application properties are reported as **distributions**, as opposed to existing characterization approaches that report average values. These metrics provide a better understanding and insight into applications properties over the entire execution.

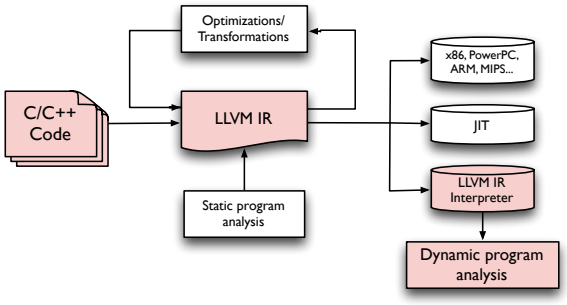


Fig. 2. Overview of the LLVM compiler infrastructure. The shadowed elements are the components from the infrastructure that are used in our characterization process.

This document is intended to provide an overview of the characterization framework, its functionality, some aspects of its implementation, and information about how to use it and install it. Section II presents a description of the LLVM compiler infrastructure—upon which the characterization framework is built—and the properties that make it a suitable platform for analysis of application properties (section III). It is followed by a description of how the application analysis is done within LLVM, and how the analysis is integrated into the LLVM infrastructure, in sections IV and V, respectively. Finally, section VI provides instructions about the installation and usage of the characterization framework, and section VII discusses some other miscellaneous aspects of the tool.

II. LLVM Compiler Infrastructure

LLVM (Low Level Virtual Machine) [11] is a compiler infrastructure that generates production quality code, in some cases comparable to that produced by `gcc` or `icc`. Its success comes from its modular design, that enables aggressive transformations and optimizations at compile-time, link-time and run-time. Although initially was developed as a research platform for dynamic optimizations, currently is the default compiler provided in Mac OS X and iOS, and is used in a number of production and research projects. It is a well-documented tool, actively maintained, widely used by both industry and academic community, and publicly available under an open source License.

Figure 2 illustrates the structure of the LLVM infrastructure. The first stage in the compilation flow consists of a front-end that translates source code to LLVM IR. There exist LLVM front-ends for a number of high-level programming languages, including C, C++, Objective-C, Java, Scala, Ada, Fortran, Python, Ruby, or Haskell. In the next stage of the compilation process, LLVM implements the different optimizations, and provides an

extensive set of libraries for static analysis of application's binaries in LLVM IR form (also known as bitcode). After the corresponding optimizations, LLVM contains different back-ends that support code generation for the most popular microarchitectures, namely, x86, PowerPC, ARM, SPARC, and PTX, which can be then executed in the corresponding CPU. Alternatively, LLVM provides a Just-In-Time compiler, that enables dynamic translation of bitcode into machine native code, and an interpreter which can directly execute LLVM bitcode. Our characterization tool extends the LLVM interpreter by adding a number of libraries that analyze the bitcode properties while it is being interpreted. The shadowed elements in Figure 2 are those components of the LLVM infrastructure that are used in our characterization framework.

III. Properties of LLVM IR

This section describes some of the features that make the LLVM IR a convenient level of abstraction for quantifying application properties.

A. Virtual Instruction Set

The LLVM Instruction Set represents a virtual architecture that captures the key operations of traditional computing systems, but avoids machine specific constraints such as physical registers, or low-level calling conventions. LLVM instructions can be broadly classified in five groups: control instructions, binary instructions, bitwise binary instructions, memory instructions and other instructions. Most LLVM operations are in three-address form and they are polymorphic, that is, a single instruction can operate on several different types of operands.

Figure 3 illustrates an example of how the LLVM virtual instruction set may abstract some of the artifacts introduced by an ISA. Figure 3(a) shows a MIPS instruction trace that loads an element from memory and multiplies it with a previously loaded element. From the ten instructions that are executed, only two correspond to the actual load and multiplication; the rest of instructions implement the arithmetic necessary to calculate the address of the memory access. This sequence of instructions is not constant across ISAs—it depends on the available instructions in the ISA—, and is also highly dependent on the level of optimization of the application. Whereas these instructions are actually executed during an application's run, they are not inherent to the algorithm, and including them in the analysis may provide a mistaken insight into the application's inherent properties. The LLVM virtual instruction set gets over this problem by defining a new instruction, the `getelementpointer` instruction, that generates the address of the element that is going to be accessed (Figure 3(b)).

```
lw    $5,8($30)
lw    $6,32($30)
mult  $5,$6
mflo  $5
lw    $6,4($30)
addu  $5,$5,$6
addu  $6,$0,$5
sll   $5,$6,0x3
lw    $6,40($30)
addu  $5,$5,$6
l.d   $f2,0($5)
mul.d $f0,$f0,$f2
```

(a)

```
%arrayidx16 = getelementptr inbounds double* %b, i64 1
%16 = load double* %arrayidx16, align 8
%mul = fmul double %0, %16
```

(b)

Fig. 3. Illustrative example of how some artifacts of the MIPS ISA, such as all the arithmetic related to address computation (a), can be abstracted with the LLVM virtual instruction set (b).

B. Single-Static Assignment Form

LLVM IR is in Single-Static Assignment (SSA) form, that is, every value is guaranteed to have only a single definition point. SSA form enables, hence, an analysis that is independent from the number of architected registers, and simplifies the iteration over def-use chains. This, in turns, facilitates tracking the last usage of data elements, which is relevant to both, parallelism and data reuse analysis.

C. Data Types

LLVM provides support for arbitrary width integers, as well as vector and SIMD data types, increasingly being used in high-performance applications. This particular feature allows us not to restrict the analysis to existing data types, but to capture exactly those data types required by the application. So, for example, if an application operates only on 11-bit integers, the analysis is done with 11-bit width data types, and not the typically defined 16-, 32- or 64-bit integers.

D. Use of Intrinsics

Intrinsics are a mechanism available in LLVM to abstract some common functionalities that have different implementations in different platforms. An example of an intrinsic is the operation unsigned multiplication with overflow, `llvm.umul.with.overflow.i64`. It is not implemented as an instruction, but as an LLVM intrinsic. In some platforms, there is a single instruction that implements this multiplication, whereas in other platforms, up to four instructions are needed. By defining it as an intrinsics, LLVM makes this operation independent from the specific target where it is going to be executed. Then, during compilation to native code, when the code for a specific target platform is generated, intrinsics

```
void Interpreter::run() {
   //Initialize ApplicationAnalyzer with the options
     specified in the command line
   ApplicationAnalyzer = new ApplicationAnalysis(
     TargetFunction, AnalyzeILP,...);
   while (!ECStack.empty()) {
      // Interpret instruction & increment the "PC".
      ExecutionContext &SF = ECStack.back();
      Instruction &I = *SF.CurInst++;
      visit(I);
      // Dispatch analyzer
      if(AnalyzeApplication)
         (ApplicationAnalyzer->*ApplicationAnalyzer->
     ApplicationAnalysisMemFn)(I,SF,Address);
   }
}
```

Fig. 4. Interpreter main loop.

are converted to the most appropriate instruction or sequence of instructions on the native architecture.

E. Vector Support

LLVM has vector support in its intermediate representation. This is an interesting feature because many contemporary architectures support vector instructions, and an increasing number of programmers rely on vectorization to speed up their computations. It is probable, hence, that applications that are going to be analyzed in the characterization framework are written with support for vector instructions. By having vector support, a broader range of applications can be analyzed without any modifications to the source code.

IV. Analysis of Applications with the LLVM Interpreter

The LLVM interpreter executed the application bitcode. It consists of a loop in which every iteration implements the execution of an instruction. Figure 4 outlines the main structure of this loop. All the instructions defined in the LLVM Instruction Set are implemented as a C++ routine, in which the content of the register or memory location corresponding to the output operand is modified according to the input operands and the operation code. The `visit(I)` call in line 10 is a call to the code that implements the instruction functionality. As shown in the code, in addition to the execution of the instruction, the interpreter initializes the application analyzer according to the options specified in the command line, and call the corresponding analysis routines.

The output of the interpreter is a text file with a summary of the statistics of the application execution, such as instruction breakdown or execution time, and, depending on the configuration flags, the distribution and cumulative distributions of ILP, TLP, etc. The details of the format of the framework output are described in [5].

A. Configuration of Application Analysis

The analysis of application's properties is controlled by a number of flags that are activated with the command-line options listed in Table I. The flags can be classified into the following categories:

Selection of the type of analysis: the framework implements a number of boolean flags that determine which of the following properties should be analyzed: instruction-level parallelism (`-analyze-ilp`), basic-block-level parallelism (`-analyze-tlp`), task partitioning (`-analyze-task-partitioning`), data reuse (`-analyze-data-temporal-reuse`), or instruction reuse (`-analyze-inst-temporal-reuse`.) When no application analysis flag is specified, only instruction mix is reported.

Ignore variables: the LLVM infrastructure allows us to specify source code lines (`-ignore-source-code-lines` flag) such that variables defined in these lines are ignored (for data dependence or data reuse analysis). If the code is written in a way that more than one variable are defined in the same source code line, and only one of them is to be ignored, in general it requires just a slight modification to the source code to separate variable definitions such that it is easy for the framework to treat them accordingly. This feature of the tool is specially useful for removing the impact of loop-carried dependences in the parallelism analysis. Identifying loop index variables in a dynamic instruction trace is in general a non-trivial task. With this feature, we can identify and remove them from the analysis by simply ignoring dependences on those variables defined in the specified lines.

Note that this feature requires the application to be compiled with debugging information, i.e., that `-g` flag should be activated during compilation to bitcode.

Restrict the analysis to a specific fraction of the code: the analysis can be applied to the entire application (`-function main`), or to a fraction of it that implements a specific function. The analysis is triggered by call to the target function, and the data is collected when the function returns. If there are several calls to the same function and it is desired to collect data from all the calls, then the code should be modified to define an auxiliary function that includes all the calls to the original target function, and make this auxiliary function the target of the analysis.

Reuse granularity: as described in section III, one of the advantages of LLVM IR is its comprehensive representation of data types. To fully exploit that characteristic of the LLVM IR, the data reuse analysis can be configured and applied to data items of different granularities (`-reuse-granularity-int-32`, `-reuse-granularity-fp-64`, etc.).

B. Analysis of Parallelism

Ideal-case parallelism is analyzed at two different granularities, at the instruction level and at the basic-block level. Parallelism is quantified by monitoring all registers and memory usage, and tracking the

TABLE I
LLI COMMAND LINE OPTIONS FOR APPLICATION ANALYSIS.

Option	Default	Description	Example
`-analyze-application`	True	Enable application characterizations	
`-function`	main	Specify function to analyze	
`-ignore-source-code-lines`	-	Specify function to analyze	`-ignore-source-code-lines={88,102}`
`-analyze-ilp`	False	Enable ILP analysis	
`-perfect-branch-prediction`	True	Perfect branch prediction for analysis of ILP	
`-inst-window-issue-size`	Infinity	Instruction window issue size (number of instructions)	
`-analyze-tlp`	False	Enable TLP analysis	
`-analyze-task-partitioning`	False	Enable analysis of inter-basic-block communication for task partitioning	
`-analyze-data-temporal-reuse`	False	Enable data temporal reuse analysis	
`-reuse-granularity-int-32`	False	Analyze reuse of 32-bit integer memory accesses	
`-reuse-granularity-int-64`	False	Analyze reuse of 64-bit integer memory accesses	
`-reuse-granularity-fp-32`	False	Analyze reuse of 32-bit floating-point memory accesses	
`-reuse-granularity-fp-64`	False	Analyze reuse of 64-bit floating-point memory accesses	
`-analyze-inst-temporal-reuse`	False	Enable instruction temporal reuse analysis	
`-print-filtered-reuse-distribution`	True	Print reuse distribution with distances rounded up to the nearest power of two	
`-cache-line-size`	4B	Specify the cache line size (B)	
`-memory-word-size`	4B	Specify the memory word size (B)	
`-nsets`	1	Specify the number of sets in a set associative cache	

inter-instructions and inter-basic-blocks read-after-write (RAW) dependences. These dependences determine the earliest cycle in which an instruction or basic-block can be executed because all its dependences have been satisfied. This information can be used to build the data dependence graph, and best-case parallelism can be estimated as the ratio of the work—total graph size— and span—length of the critical path [7]. Since the dependence graph contains information about how many instructions/ basic-blocks can be executed in every cycle, more detailed statistics about parallelism can be reported, such as the distribution of parallelism over time, or the cumulative distribution. Figures 5(a) and 5(b) show examples of the reported parallelism. In the case of instruction-level parallelism, the data can be further broken down into instruction types, and per-instruction-type distributions can be also obtained (Figure 5(c)).

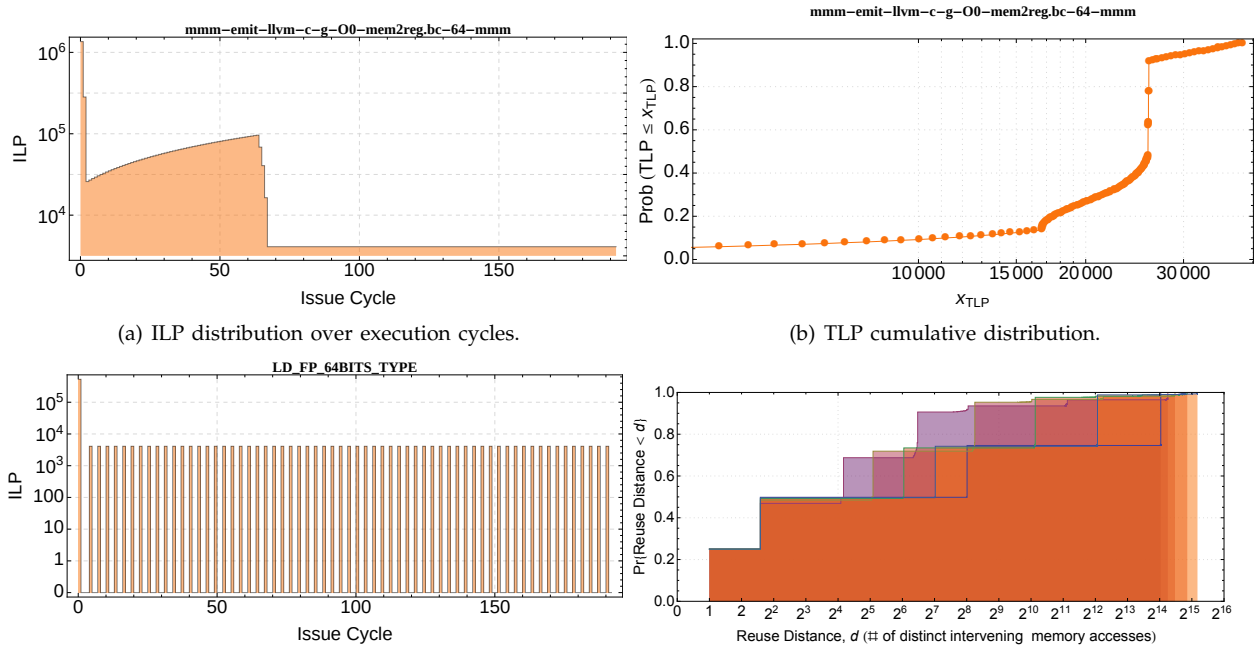
C. Analysis of Locality

Locality has traditionally been defined in qualitative terms [10]. Several studies attempt to define a single-value metric that quantifies locality [14, 18]. These metrics, however, do not provide enough insight into how the actual memory access pattern of the application is, or how these patterns actually impact applications' performance. We use the reuse distance distribution as the metric to quantify locality, where the reuse distance is defined as the number of distinct intervening memory access between two consecutive accesses to the same memory location. The first algorithm to calculate reuse distance [13] was based on a stack data structure, such that whenever a new data element was referenced (memory page, cache block, or memory word, depending on the granularity at which the analysis is being done), this element is located on top of the stack, and the rest of elements in the stack are down-shifted. This reuse distance analysis technique is also known as *LRU stack distance*, as it models the cache behavior of a fully-associative cache memory with least-recently used replacement policy. Given the inefficiency and complexity of early proposed stack algorithms, several optimizations of the algorithm have been proposed [1, 3]. The characterization tool implements the algorithm described in [16], which uses a splay tree to efficiently calculate reuse distances. The result of such algorithm is a trace of stack distances that can be represented as an histogram or, equivalently, cumulative distribution, and this reuse distance cumulative distribution can be used to directly predict the hit rate (or miss rate) for every cache size, assuming LRU replacement policy and fully associativity. Figure 5(d) shows an example of such cumulative distribution.

V. INTEGRATION OF THE APPLICATION ANALYSIS TOOL INTO THE LLVM INFRASTRUCTURE

As mentioned in Section II, one of the benefits of using LLVM is its modular design. A premise during the design of the characterization tool was to minimize the LLVM source code that had to be modified; that would make the characterization tool less sensitive to the continuously changing LLVM source code, and would facilitate its integration and installation. The tool, hence, has been implemented as another LLVM module or library whose routines are invoked by the interpreter when required. Figure 6 describes the interaction of the LLVM source code and the files that have been added. The only LLVM source file that is modified is the interpreter—`Interpreter/Execution.cpp`—, which has been extended to parse the new command line options that specify which analyses must be done, and to invoke the application analyzer —implemented in `Support/ApplicationAnalysis.cpp`—when the analysis has been activated in the interpreter invocation command line.

Parallelism analysis is implemented in `ParallelismAnalysis.cpp`, which implements general routines, and `ILP.cpp` and `TLP.cpp`, which implement aspect of the analysis that are particular to each kind of parallelism. Similarly, locality routines are implemented



(a) ILP distribution over execution cycles.

(b) TLP cumulative distribution.

(c) ILP of load operations

(d) Cumulative reuse distributions

Fig. 5. Examples of application's properties reported by the characterization framework.

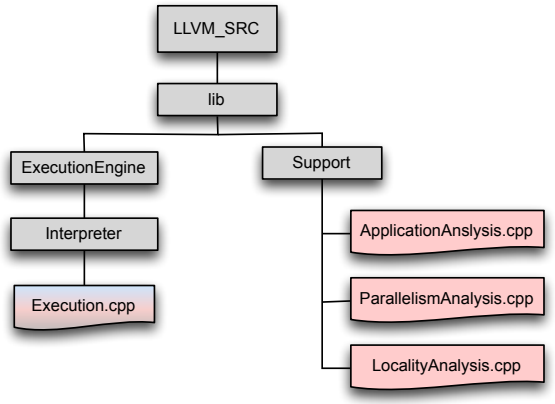


Fig. 6. Integration and interaction of application analysis source code into the LLVM infrastructure. Rectangles represent directories of the original LLVM source code infrastructure, the red documents are the new files added to implement the application analysis, and the red-blue document is an LLVM original file that has been modified.

in `LocalityAnalysis.cpp`, `StackReuseDistance.cpp`, and `OptReuseDistance.cpp`.

VI. Installation and Use

The characterization tool is provided as:

- A Mathematica Workbench project, called `ApplicationCharacterization`, which contains a notebook with the user interface to configure the analysis, and a package that implements the routines for post-processing the data and generating the results (components 1, 3, and 4 in Figure 1(a)). If used, it must be located in the directory `MATHEMATICA`, as shown in Figure 7(b).
- A patch to the LLVM compiler infrastructure that adds the support for the application analysis to the LLVM infrastructure (component 2 in Figure 1(a)). The patch and installation instructions are provided in `llvm-app-analysis.tar.gz`.

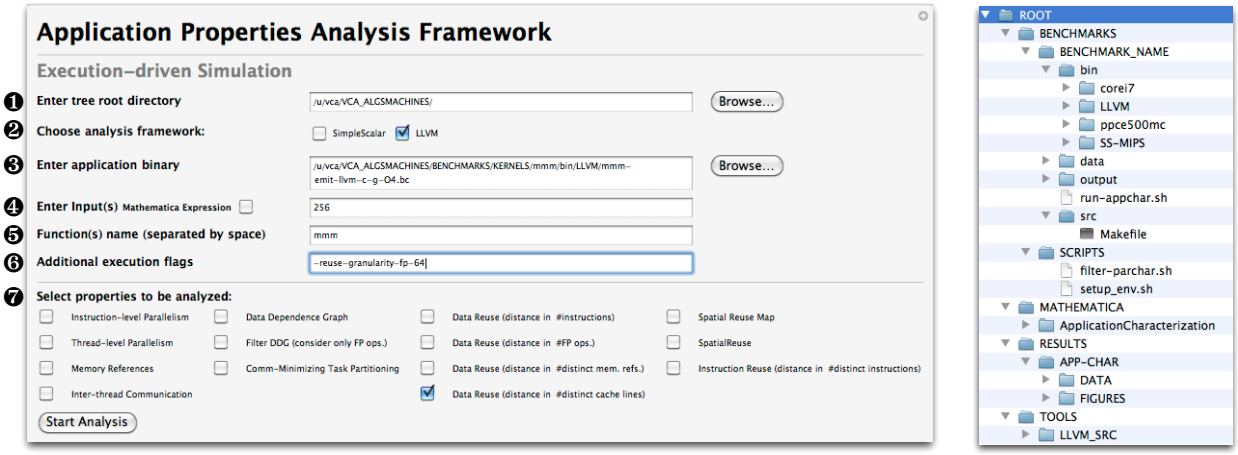
Figure 7 illustrates the Mathematica interface and the directory structure in which the tool relies for its execution. It requests the root of the underlying directory structure (Figure 7(b)), the path to the application bitcode[1], the input of the applications execution, as well as the function name—if any—to be analyzed. The check boxes allow the user to select which of the available analysis should be executed, and other configuration options, should be entered in the *Additional execution flags* field.

VII. Some Comments about the Tool

A. Tool Scalability and Execution Time

Execution of applications in interpreted mode incurs in performance losses. For large applications in which performance is critical even in native execution, the slowdown due to the execution on a simulator and the overhead added by the data flow analysis, might become an inhibitor of this technique. Therefore, during the tool design process, special attention was put in the efficient implementation of data-flow-track algorithms and the election of the most appropriate data structures depending on their size and access pattern.

[1]The characterization framework assumes applications have been compiled into LLVM bitcode, and the bitcode is located in the corresponding `bin` directory. The command line for compiling applications is: `clang -emit-llvm -c -g -O0 *.c -o *.bc`



(a) (b)

Fig. 7. (a) Mathematica interface to enter application analysis configuration and launch the analysis (b) Directory structure on which the characterization framework relies.

B. Modifications to Applications

In some cases, running applications through the characterization framework might not be straightforward, either because applications cannot be compiled to bitcode, because when the bitcode is executed there is an error in the LLVM interpreter, or because the properties reported in the analysis do not match the expected properties. These cases are typically solved by doing some minor modifications to the source code. These modifications do not change the functionality of the code, but simply help the compiler to generate bitcode that contains all the necessary features for the analysis. In general, is it recommended that the source code does not use complex data structures. Also, if the code is compiled to bitcode with the optimization flag `-O4`, it is possible that functions are inlined and, hence, it is not possible to track properties *per function*. In these cases, it is recommended to add `__attribute__ ((noinline))` to the function to be analyzed. Finally, If the execution process fails, running the interpreter with the `-debug` flag will print out the instruction execution trace, and some other debugging information, what will help to find the part of the application is creating the problem.

References

[1] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. *SIGPLAN Not.*, 38:37–43, June 2002.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb. 2002.

[3] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19:353–357, 1975.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25:13–25, June 1997.

[5] V. Caparrós Cabezas. File format description of analysis framework output. DOME - TN1, 2012.

[6] V. Caparrós Cabezas and P. Stanley-Marbell. Parallelism and data movement characterization of contemporary application classes. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 95–104, New York, NY, USA, 2011. ACM.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc. Balance principles for algorithm-architecture co-design, 2011.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[12] M. A. Postiff *et al.* The limits of instruction level parallelism in SPEC95 applications. *SIGARCH Comput. Archit. News*, 27(1):31–34, 1999.

[13] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78 –117, 1970.

[14] R. Murphy, A. Rodrigues, P. Kogge, and K. Underwood. The implications of working set analysis on supercomputing memory hierarchy design. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 332–340, New York, NY, USA, 2005. ACM.

[15] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[16] F. Olken. *Efficient methods for calculating the success function of fixed space replacement policies*. 1982.

[17] D. W. Wall. Limits of instruction-level parallelism. *SIGARCH Comput. Archit. News*, 19(2):176–188, Apr. 1991.

[18] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 50–, Washington, DC, USA, 2005. IEEE Computer Society.

[19] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[20] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.*, 31:20:1–20:39, August 2009.