# Automated Energy/Performance Macromodeling of Embedded Software

Anish Muttreja[†], Anand Raghunathan[‡], Srivaths Ravi[‡] and Niraj K. Jha[†]
[†]Dept. of Electrical Engineering, Princeton University, NJ 08544
[‡]NEC Labs, Princeton, NJ 08540

## Abstract

*Efficient energy and performance estimation of embedded software is a critical part of any system-level design flow. Macromodeling based estimation is an attempt to speed up estimation by exploiting reuse that is inherent in the design process. Macromodeling involves pre-characterizing reusable software components to construct high-level models, which express the execution time or energy consumption of a sub-program as a function of suitable parameters. During simulation, macromodels can be used instead of detailed hardware models, resulting in orders of magnitude simulation speedup. However, in order to realize this potential, significant challenges need to be overcome in both the generation and use of macromodels— including how to identify the parameters to be used in the macromodel, how to define the template function to which the macromodel is fitted,* etc. *This paper presents an automatic methodology to perform characterization-based high-level software macromodeling, which addresses the aforementioned issues. Given a sub-program to be macromodeled for execution time and/or energy consumption, the proposed methodology automates the steps of parameter identification, data collection through detailed simulation, macromodel template selection, and fitting. We propose a novel technique to identify potential macromodel parameters and perform data collection, which draws from the concept of* **data structure serialization** *used in distributed programming. We utilize* **symbolic regression techniques** *to concurrently filter out irrelevant macromodel parameters, construct a macromodel function, and derive the optimal coefficient values to minimize fitting error. Experiments with several realistic benchmarks suggest that the proposed methodology improves estimation accuracy and enables wide applicability of macromodeling to complex embedded software, while realizing its potential for estimation speedup. We describe a case study of how macromodeling can be used to rapidly explore algorithm-level energy tradeoffs, for the* `zlib` *data compression library.*

## Categories and Subject Descriptors

I.6.5 [**Computing Methodologies**]: Simulation and Modeling - *Model development - Modeling methodologies*; D.2.8 [**Software**]: Software Engineering - *Metrics, Performance measures*; C.4 [**Computer Systems Organization** ]: Performance of Systems- *Modeling Techniques*

## General Terms

Design, Measurement

## Keywords

Data Serialization, Embedded Software, Genetic Programming, Macromodeling, Symbolic Regression

## 1. INTRODUCTION

Efficient performance and energy estimation are fundamental concerns in the design of embedded software. Simulating the execution of embedded software on models of the underlying processor platform is the most widely used approach for performance and energy estimation. While simulation efficiency has been the subject of significant research effort, rapid growth in the complexity of embedded software (the number of lines of code in a typical embedded application is estimated to double every 10 to 12 months on an average, *i.e.*, even faster than Moore's law) implies that efficient performance/energy estimation for embedded software will remain a challenge for the foreseeable future.

Our work is based on the observation that large embedded software programs are rarely written from scratch — reliable design, subject to stringent design turnaround time and design cost constraints, mandates substantial *reuse*. An analysis of the dynamic execution traces of embedded programs reveals that a large fraction of the time consumption arises from reused software components (including embedded operating systems, middleware, run-time libraries, domain-specific algorithm libraries, *etc.*). As an example, our experiments with the popular compression utility *gzip*, showed that, on an average, 90% of *gzip's* execution time is spent in calls to the *gzip* library[1] package, 8% in calls to the standard C library functions, and only 2% in code specific to the *gzip* program, or what is frequently known as "glue code." It is hence not surprising that reusable software modules account for a major fraction of simulation or estimation effort.

*It is natural to wonder whether reuse, which saves significant design effort, can also be exploited to reduce estimation effort.* Characterization-based macromodeling takes a step in the above direction by enabling the extraction of fast, higher level models of reusable software components, based on pre-characterization using more detailed, slower models. The effort expended in the construction of macromodels for a software module is amortized over the large number of applications of the macromodel when the module is simulated in the context of all the programs that include it.

The rest of this paper is organized as follows. We describe the contributions of this paper in Section 1.1 and discuss related work in Section 1.2. In Section 2, we identify the major challenges involved in macromodeling. Section 3 describes in detail how the proposed macromodel generation methodology overcomes the identified challenges. Our implementation and experimental results are presented in Section 4, and conclusions in Section 5.

### 1.1 Paper Contributions

*The complexity of modern embedded software poses significant challenges for both the generation and use of macromodels.* In this work, we identify key limitations of the state-of-the-art in software macromodeling. Notably, significant manual effort is required from the software designer towards the identification of suitable parameters, and a template function on which the macromodel is based. Addressing these problems, while maintaining sufficient generality in order to handle a wide range of embedded software programs is quite challenging. We propose a methodology to automate the critical steps of parameter identification, data collection through accurate simulation or measurement, and construction of the macromodel function while simultaneously optimizing the values of macromodel coefficients for achieving the best fit. Our work draws inspiration from concepts presented in the fields of distributed programming (automatic data structure serialization), and recent advances in statistical data analysis (symbolic regression). We also demonstrate the practical application of macromodeling to software libraries of significant complexity.

[1]The gzip library is used by software packages that internally use the *gzip* compression engine.

```
extern void bg_compute_scc ( /*BGRAPHPTR bgr;*/);
struct bgraph {
        int first, last;
        int no_vertices;
        BOOLEAN *active;// v is active when
        //active[v] is T
        LISTPTR *vlist; // adj list representation
        LISTPTR *scc;   // records the strongly
        //connected components
        int *v2scc;                 //records the scc
        //corresponding to vertex
        int scc_valid;    // 1 when scc
        //corresponds to vlist, else 0
};
typedef struct list LIST;
typedef struct list *LISTPTR;
typedef long LISTOBJ;
struct list {     // linked list structure
        LISTPTR next;   // doubly linked list
        LISTPTR prev;
        LISTOBJ o;      // object in the list
};
```

**Figure 1: Function `bg_compute_scc` and associated data structures**

## 1.2 Related Work

We discuss related work in the areas of macromodeling for hardware power simulation, efficient software performance and energy estimation, and fast instruction set simulation. Macromodels for register-transfer level (RTL) components can be constructed through characterization of their logic-level hardware models and have been used extensively in RTL power estimation [1, 2]. Techniques to speed up cycle-accurate instruction set simulation have received significant attention. Instruction-set simulation can be accelerated with little or no loss of accuracy using compiled simulation [3], combining compiled and interpreted simulation [4], or by optimizing the implementation of different functions such as instruction fetch and decode [5] in the instruction set simulator (ISS). Hybrid simulation techniques for energy estimation were proposed in [6]. Delay and energy caching (reusing the execution time and energy consumption results from previous simulations of a program segment) are used to speed up estimation in [7, 8].

An alternative approach to embedded software power analysis is to use cycle-accurate and structure-aware architecture simulators, which can identify the architectural blocks activated in each cycle during a program's execution, and record the stream of inputs seen by them [9, 10]. Software macromodeling at the granularity of functions or sub-programs was explored in [11, 12], demonstrating that orders of magnitude speedup in estimation time could be obtained, while maintaining high estimation accuracy. Performance characterization of library functions using statistical and semantic properties of function arguments was recently presented in [13]. In summary, the importance of embedded software performance and energy estimation has fueled significant research effort but macromodeling for software sub-programs of arbitrary complexity has remained a relatively unexplored area, and many important issues have not been addressed. To the best of our knowledge, this is the first work to automate all the key steps in macromodel generation and demonstrate the applicability of fully automatic macromodeling to software programs of realistic complexity.

## 2. MOTIVATION

In this section, we describe the key challenges involved in macromodel generation for complex software programs, and illustrate them through the task of constructing an energy macromodel for the `bg_compute_scc` function taken from a commercial graph data structure library. The C prototype of function `bg_compute_scc` is shown in Figure 1, along with a description of its input data structures. The `bgraph` structure contains various dynamically allocated fields, including an adjacency list representation of the graph's connectivity, and fields to store the identified strongly connected components (SCCs). In addition to the software implementation of the graph data structure library and several testbenches that exercise its functions, we are given a cross-compilation tool chain for the target StrongARM based embedded system, as well as a cycle-accurate ISS that reports energy consumption [14, 15]. Any automated approach to generating a macromodel needs to address the following key challenges:

- *Selection of macromodel parameters:* In general, macromodel parameters, which are the independent variables used in the macromodel, can include the size or value of any field nested arbitrarily deep within the input or output data structures. The number of candidate parameters can be very large even for simple software functions. However, an efficient and robust macromodel must include only relevant parameters that have an actual impact on energy consumption. For the `bg_compute_scc` function, if we consider the values of all nested fields of scalar data types and the sizes of all nested fields of complex types, we can identify $2n+e+s+9$ potential candidates for macromodel parameters for a graph with $n$ vertices, $e$ edges, and $s$ SCCs. The number of possible relevant subsets of parameters is $2^{2n+e+s+9}$. While in some cases, human understanding and insight may reveal that only a small subset of parameters may largely determine the execution time or energy consumption, an automatic tool processing the source code does not have the luxury of human insight.

- *Data collection:* Once a candidate set of macromodel parameters is identified, characterization experiments must be performed to obtain values of the candidate macromodel parameters and the corresponding value of the dependent variable (energy or execution time) for numerous execution instances. Capturing the macromodel parameter values requires run-time tracing of the size of dynamically-created data structures as well as values of nested atomic fields. In practice, this is not a simple task — the number of levels of pointer traversals that have to be performed to access all scalar fields may vary dynamically, and conventional size computation utilities (such as `sizeof` in the C programming language) do not perform pointer traversal, *i.e.*, they do not include the size of objects pointed to by fields in the given object.

- *Macromodel function construction:* Given the data gathered from characterization, determining a suitable function to approximate the collected data can be a daunting task. The search space of functions is highly intractable (infinite in the case of real-valued functions). Conventional approaches to macromodeling circumvent this problem by requiring the designer to manually specify a macromodel template. While various templates have been suggested, in [12] for example, template identification is in practice an *ad hoc* and tedious process that demands a detailed understanding of the target function.

We present techniques using type analysis, data serialization concepts and symbolic regression to overcome these challenges, making it possible to significantly extend the applicability of macromodeling to complex software, while greatly simplifying macromodel construction and minimizing the need for human intervention.

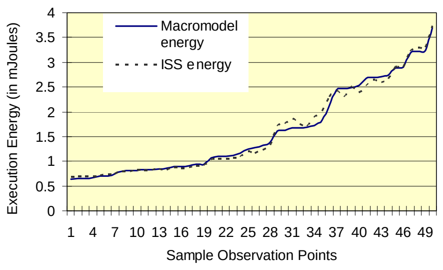

**Figure 2: Energy estimates from macromodeling and instruction-level simulation for `bg_compute_scc`**

To illustrate the utility of our methodology, we used it to construct an energy consumption macromodel for the `bg_compute_scc` function shown in Figure 1. The resulting macromodel equation, which relates energy consumption to the size[2] of the input argument *bgr* and values of its member fields, is as follows:

[2]We define our notion of *size* in Section 3.1

$Energy = (5.83E-6) * last * size(bgr) + (0.5934) * no\_vertices - (0.576) * last + (3.625E-4) * size(bgr)$.

A model in terms of function arguments, like the one shown above, also has the additional advantage of being well-suited to automated macromodel application within a larger estimation framework, because the model parameters should be readily available in any software simulator. A comparison of the energy estimates from the use of the macromodel *vs.* the energy estimates from instruction-level simulation for various input instances, as shown in Figure 2, shows them to be in close agreement with instruction-level estimates, with an average estimation error of just $0.7\%$.

## 3. AUTOMATIC MACROMODELING METHODOLOGY

Figure 3 presents an overview of the proposed macromodeling methodology. Starting with the source code for the target function to be macromodeled, and a testbench that thoroughly exercises the target function over a wide range of input instances, the methodology consists of a sequence of steps that culminates in the generation of macromodels which approximate the energy consumption or execution time of the function. Two parallel compilation and execution flows are used to collect the data necessary to construct the macromodel. First, the source code is subject to parsing and type analysis, based on which our tool automatically generates data structure traversal and serialization routines and instruments the source code to invoke them at appropriate locations. The instrumented source code, traversal and serialization routines, and testbench are compiled and executed (any functionally accurate execution environment suffices for this step). During execution, the traversal and serialization routines dynamically enumerate and collect the values of candidate macromodel parameters. Cross-compilation and instruction-level simulation of the uninstrumented target source code and testbench is used to obtain the energy consumption and execution time for each execution instance of the target function. The collected instance-by-instance profile database, which contains values for all the independent and dependent variables, is then fed to the symbolic regression engine to produce the macromodel.

The rest of this section describes the key steps of our methodology, which are highlighted as shaded rectangles in Figure 3.

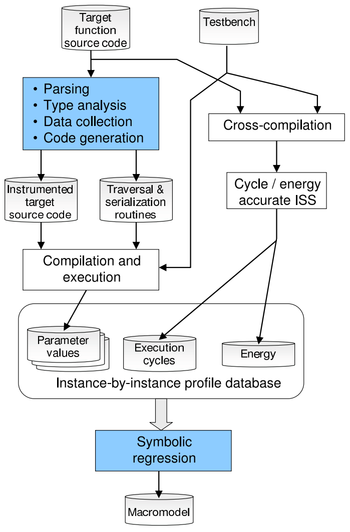
Target function source code
Testbench
• Parsing
• Type analysis
• Data collection
• Code generation
Cross-compilation
Cycle / energy accurate ISS
Instrumented target source code
Traversal & serialization routines
Compilation and execution
Parameter values
Execution cycles
Energy
Instance-by-instance profile database
Symbolic regression
Macromodel

**Figure 3: Overview of the proposed automatic macromodeling methodology**

### 3.1 Data Collection

Our data collection tool parses the input C files[3] to collect information about data types and function arguments in the program, which is used to suitably instrument the input program.

[3]Although our specific implementation is for the C programming language, the concepts presented in this work are fairly language-independent.

We use argument sizes and values of the input and output data structures of the target function, as well as their nested fields, as model parameters.

We define *argument size* of a data structure as the number of bytes it would occupy if it were serialized. Serialization is the process of converting structured data to serial byte streams for the purpose of storage or transmission, as in a remote procedure call [16].

We use *type analysis* to automatically generate code that computes argument sizes. In compiler theory [17], two types of data types are identified: *basic types* (*e.g,* `int`, `char`, `float`) and *type constructors* (*e.g,* pointers, arrays, and structures). Our object size calculations utilize rules associated with each basic type and type constructor. The size of basic types can be directly obtained using language facilities. The size of a structure is the sum of all nested fields. Pointers are recursively traversed using indirection until a non-pointer type is obtained, whose size is then taken as the size of the pointer. Array sizes can be calculated similarly but require knowledge of array bounds at runtime. While C implementations do not in general maintain array bounds, C functions that have array arguments usually also include other arguments specifying array bounds.

Callee function argument sizes are computed dynamically by code inserted in the caller function that calls the target function, immediately before and after the call. The framework described above enables run-time calculation of object size and other interesting information. For example, the size of a linked list object would be calculated as the sum of the sizes of all elements of the linked list. As a more complex example, consider the `bgraph` structure shown in Figure 1. Most macromodel templates for `bg_compute_scc` would require data about the number of vertices, $n$, and number of edges, $e$, in the graph. From the value of field `no_vertices`, $n$ can be obtained directly. Calculating $e$ requires recognizing that `vlist` (the graph's adjacency list) is actually an array of size $no\_vertices$ of *LISTPTR* objects. Hence, the size of the `vlist` field ends up serving as a good estimate of $e$.

### 3.2 Macromodel Construction Using Symbolic Regression

The data collected through characterization experiments should be used to construct a macromodel relating the target function's energy or execution time to a subset of the potential macromodel parameters. We perform this critical step through the use of *symbolic regression*, which was first introduced as an application that combined the fields of statistical data analysis and genetic programming (GP) by Koza [18]. GP is used to evolve formulae containing the identified model parameters and a chosen set of mathematical operators. Based on extensive experimentation, we found the set $\mathcal{F} = \{+, -, \times, /, x^2, x^3, \text{and} \log(x)\}$ of operators to be quite adequate for our modeling needs.

We used an extended form of Koza's symbolic regression technique, called Hybrid GP (HGP) [19], to increase the numerical robustness of symbolic regression. HGP extends Koza's symbolic regression by introducing weights for all additive terms in the genetically derived regression formula. Classical linear regression is used to tune the weights.

## 4. IMPLEMENTATION AND RESULTS

In this section, we discuss our implementation and present experimental results demonstrating the benefits of the proposed methodology.

The instrumentation and data collection steps in our methodology were implemented using a YACC based parser [20] and several PERL scripts. Our implementation of symbolic regression is based on the GP kernel `gpc++` [21] and libraries for symbolic [22] and numerical manipulation [23].

We conducted several experiments using a variety of benchmark software programs to demonstrate the utility of our automatic macromodeling framework. Table 1 shows how the macromodels obtained using our framework perform compared to execution times and energy consumption values obtained through a combination of extensive simulations and measurements from real implementations. The benchmarks have been given descriptive names to indicate their function. For each benchmark, a sample set of 500 input instances (data sets) was used to develop the macromodel. The error associated with a macromodel is defined as the root mean square (RMS) deviation from observed values (obtained through instruction-set simulation or measurement), taken over the entire sample set. The symbolic regression tool was programmed to terminate after fifty GP generations or when the error dropped to less than 1%, whichever occurred sooner.

**Table 1: Macromodeling examples**

| Example | Source | Perf. Error | Energy Error |
|---|---|---|---|
| convex hull | 2dch [24] | 0.2% | 0.3% |
| bg_compute_scc | NEC Labs | 0.3% | 0.7% |
| shortest path | wnlib [25] | 0.3% | 0.5% |
| linked list sort | NEC Labs | 0.1% | 0.1% |
| simplex | wnlib | 0.8% | 0.5% |
| bipartite matching | bipm [26] | 0.8% | 0.9% |
| strncat | glibc | 0.9% | 0.2% |
| qsort | glibc | 0.1% | 0.4% |
| malloc | glibc | 0.4% | 0.4% |
| bsearch | glibc | 1.3% | 0.3% |
| pow | glibc | 0.3% | 0.7% |

We chose the `SimIt-ARM-1.1` cycle-accurate ARM ISS [27] as our measurement platform because of its high simulation speed. The execution time of a code segment was determined as the difference in execution times of two versions of the benchmark, one with the execution of the target function enabled, and the other with it disabled. To compute energy consumption, we extended `SimIt-ARM-1.1`, in a manner similar to the tool discussed in [15], to report processor and memory energy estimates, using the instruction and memory power models reported in [14] and [28], respectively.

### 4.1 Case Study: Energy Tradeoffs during Lossless Data Compression

In this section, we explore the use of macromodels in making algorithmic tradeoffs using the `zlib` [29] compression library. `zlib` can be embedded into any software application in order to perform lossless data compression. The *compress2*() function provided by `zlib`, whose interface is given by *int compress2* (*Bytef *dest, uLongf *destLen, uLong sourceLen, int level*), allows the user to vary the computational effort expended in compression by using the `level` function argument that takes values from zero (no compression) to nine (maximum compression).

We developed a macromodel for the energy consumed by the `compress2` function using the proposed methodology, and used it to study the tradeoff between energy consumption and the actual compression ratio achieved, for various values of the `level` parameter, over 300 files of various types ranging in size from 1 byte to 1 MB. It can be seen from the results of this experiment in Figure 4 that the average energy consumption increases monotonically with `level` but the compression ratio does not, indicating that not all compression levels are Pareto-optimal in terms of the above metrics. The figure also shows that macromodel estimates are in close agreement with energy estimates obtained using `SimIt-ARM`. Furthermore, the macromodel based approach has the same relative trend as the simulation based estimates, which makes it suitable for high-level design space exploration. The advantage of macromodeling is evident from the fact that estimation using the macromodel for all the input samples required less than a minute, while the ISS took over a day to complete.

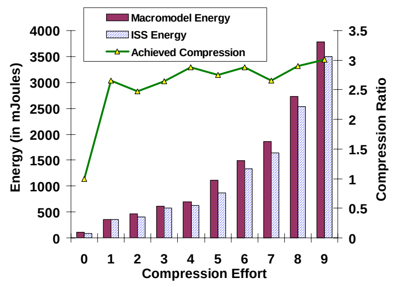

**Figure 4: Using macromodeling to explore compression *vs.* energy tradeoffs**

## 5. CONCLUSIONS

We presented a systematic methodology to automate the generation of energy and performance macromodels for embedded software. The proposed methodology radically simplifies macromodel construction, while expanding its applicability to complex embedded software. Furthermore, the use of properties of program data structures, including function arguments, as model parameters simplifies macromodel use, enabling usage in conjunction with any simulation environment. For example, macromodels could be integrated into an instruction-level simulation environment, so that some parts of the code are handled using macromodels, while glue code or parts that are difficult to macromodel are simulated using conventional techniques.

## 6. REFERENCES

[1] J. Rabaey and M. Pedram (Editors), *Low Power Design Methodologies*. Kluwer Academic Publishers, Norwell, MA, 1996.

[2] A. Raghunathan, N. K. Jha, and S. Dey, *High-level Power Analysis and Optimization*. Kluwer Academic Publishers, Norwell, MA, 1998.

[3] V. Zivojnvic, S. Tjiang, and H. Meyr, "Compiled simulation of programmable DSP architectures," in *Proc. IEEE Wkshp. VLSI Signal Processing*, May 1995, pp. 73–80.

[4] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *Proc. ACM/IEEE Design Automation Conf.*, June 2002, pp. 22–27.

[5] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt, "An efficient retargetable framework for instruction-set simulation," in *Proc. IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign & System Synthesis*, Oct. 2003, pp. 13–18.

[6] V. S. P. Rapaka and D. Marculsecu, "Pre-charcterization free, efficient power/performance analysis of embedded and general purpose software applications," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2003, pp. 504–509.

[7] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, "Software timing analysis using HW/SW cosimulation and instruction set simulator," in *Proc. Int. Wkshp. Hardware-Software Codesign*, Mar. 1998, pp. 65–70.

[8] M. Lajolo, A. Raghunathan, and S. Dey, "Efficient power co-estimation techniques for system-on-chip design," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2000, pp. 27–34.

[9] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimization,," in *Proc. Int. Symp. Computer Architecture*, June 2000, pp. 83–94.

[10] W. Ye, N. VijayKrishnan, M. Kandemir, and M. Irwin, "The design and use of SimplePower: A cycle accurate energy estimation tool," in *Proc. ACM/IEEE Design Automation Conf.*, June 2000, pp. 340–345.

[11] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak, "Function-level power estimation methodology for microprocessors," in *Proc. ACM/IEEE Design Automation Conf.*, June 2000, pp. 810–813.

[12] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha, "High-level energy macro-modeling of embedded software," *IEEE Trans. Computer-Aided Design*, vol. 21, pp. 1037–1050, Sept. 2002.

[13] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "Library function timing characterization for source-level analysis," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2003, pp. 1132–1133.

[14] A. Sinha and A. P. Chandrakasan, "JouleTrack - A web based tool for software energy profiling," in *Proc. ACM/IEEE Design Automation Conf.*, June 2001, pp. 220–225.

[15] T. K. Tan, A. Raghunathan, and N. K. Jha, "A simulation framework for energy-consumption analysis of OS-driven embedded applications," *IEEE Trans. Computer-Aided Design*, vol. 22, pp. 1284–1294, Sept. 2003.

[16] J. Bloomer, emphPower Programming with RPC. O'Reilly and Associates, Inc., Sebastopol, CA, 1992.

[17] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley Publishing Company, Reading MA, 1986.

[18] J. R. Koza, *On the Programming of Computers by Natural Selection*. The MIT Press, Cambridge MA, 1992.

[19] G. R. Raidl, "A hybrid GP approach for numerically robust symbolic regression," in *Proc. Annual Conf. Genetic Programming*, July 1998, pp. 323–328.

[20] P. Long, "Metre v2.3." [Online]. Available: http://www.lysator.liu.se/c/metre-v2-3.html

[21] A. Fraser and T. Weinbrenner, "The Genetic Programming Kernel." [Online]. Available: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/weinbenner/gp.html

[22] C. Bauer, A. Frink, and R. Freckel, "GiNaC is not a CAS," http://www.ginac.de.

[23] The GNU Free Software Foundation, "The GNU Scientific Library," http://www.gnu.org/software/gsl/.

[24] K. Clarkson, "2dch.c." [Online]. Available: http://www.math.niu.edu/~rusin/known-math/96/convhul

[25] B. Chapman and W. Naylor, "wnlib." [Online]. Available: http://www.willnaylor.com/wnlib.html

[26] R. Anderson, "Bipm." [Online]. Available: http://www.cs.sunysb.edu/~algorith/implement/bipm/distrib/

[27] W. Qin, "The SimIt-ARM simulator." [Online]. Available: http://www.ee.princeton.edu/~wqin/armsim.htm

[28] J. Flinn, K. I. Farkas, and J. Anderson, "Power and energy characterization of the ITSY pocket computer (version 1.5)," Compaq Western Research Laboratory, Tech. Rep., Feb. 2000.

[29] J-L Gailly and M. Adler, "zlib-1.1.14." [Online]. Available: http://www.gzip.org/zlib/