Hybrid Simulation for Embedded Software Energy Estimation

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

Abstract

Software energy estimation is a critical step in the design of energy-efficient embedded systems. Instruction-level simulation techniques, despite several advances, remain too slow for iterative use in system-level exploration. In this paper, we propose a methodology called *hybrid simulation*, which combines instruction set simulation with selective native execution (execution of some parts of the program directly on the simulation host computer), thereby overcoming the disadvantages of instruction-level simulation (low speed) and pure native execution (estimation accuracy, inapplicability to target-dependent code), while exploiting their advantages. Previously developed techniques for software energy macromodeling are utilized to estimate energy consumption for natively executed sub-programs. We identify and address the main challenges involved in hybrid simulation, and present an automatic tool flow for it, which analyzes a given program and selects functions for native execution in order to achieve maximum estimation efficiency while limiting estimation error. We have applied the proposed hybrid simulation methodology to a variety of embedded software programs, resulting in an average speed-up of 70% and estimation error of at most 6%, compared to one of the fastest publicly-available instruction set simulators.

Categories and Subject Descriptors: I.6.8 Computing Methodologies: - *Types of Simulation - Combined*

General Terms: Design, Measurement

Keywords: Embedded Software, Energy Estimation, Energy Macromodels, Hybrid Simulation, Pointer Analysis

1. INTRODUCTION

The most widely used technique for embedded system energy estimation is to simulate the execution of embedded software on a model of the underlying hardware platform. Due to rapid growth in the complexity of embedded software (average software content is estimated to double every 10 to 12 months, *i.e.*, faster than Moore's law [1]), software simulation is becoming the bottleneck in efficient system simulation. In spite of significant research effort over the last decade, software simulation is still very time-consuming for systems of realistic complexity, limiting the scope for architectural optimization and design space exploration.

We propose hybrid simulation, which combines instruction set simulation with native execution of judiciously selected parts of the program, as an approach to significantly improve energy estimation efficiency. Native execution, which refers to execution of a program directly on a simulation host computer, is commonly used for functional verification (without timing or energy estimation) during the early phase of the software design process. Native execution is typically much (one or more orders of magnitude) faster than instruction set simulation. However, it cannot be applied to programs that contain target-dependent code (*e.g.*, assembly code). Also, native execution by itself cannot provide estimates of energy or execution time.

Our work attempts to combine the advantages of instruction set simulation, such as applicability to arbitrary programs and accurate estimation, with the speed of native execution. In order to realize this objective, several challenges need to addressed. The basic challenges include how to perform energy estimation for code that is natively executed during simulation, and how to synchronize and communicate between the two domains to maintain functional correctness. These are significant challenges in the presence of pointers, complex data structures, and dynamic memory allocation. In addition, natively executed code can lead to estimation errors, as well as simulation overheads due to control/data transfers between domains and due to the evaluation of energy macromodels. Hence, in order to realize the potential benefits of hybrid simulation (maximal simulation speed with minimal estimation error), it is necessary to optimize the partitioning of the program between the instruction set simulator (ISS) and native execution domains while considering the tradeoffs involved. Finally, the steps involved in hybrid simulation should be automated so that little or no additional manual effort is involved.

1.1 Paper Contributions and Related Work

Next, we review related work in the areas of instruction set simulation and software macromodeling.

A large body of work deals with improving the efficiency of instruction set simulation, a detailed review of which is outside the scope of this paper. Some notable developments include compiled simulation [2], techniques that combine compiled and interpreted simulation [3, 4], and automatic generation of optimized ISS components for steps such as instruction execution and instruction decoding [5, 6]. The proposed work is largely complementary to, and can be combined with, the above techniques for instruction set simulation. Techniques for profitably identifying regions of a program that are selectively simulated in detail, while fast-forwarding over other portions have also been reported [7, 8].

In the proposed technique, unlike the aforementioned efforts, fast-forwarded code is directly compiled to host binary and executed without any details about the target processor state being maintained, which leads us to expect considerable speed-ups. However, our approach requires the use of macromodels to estimate energy consumption in natively executed code.

Previous work on software macromodeling [9, 10] has mostly focused on generation of macromodels for pre-identified parts of a programs, without addressing the question of how to identify parts of a program that may be profitably macromodeled. Moreover, a major drawback of native execution, its inapplicability to target-dependent software, has also not been addressed. Our work addresses these issues by providing a generally applicable methodology to leverage native execution and macromodels. The two extreme scenarios of hybrid simulation, where the entire application is simulated in the ISS, and the entire application is natively executed, correspond to the current state-of-the-art. Our contribution is in enabling the spectrum of possibilities that lie in between these two extremes.

2. MOTIVATION

In this section, we present the key challenges involved in hybrid simulation, which can be broadly classified into the following two types: challenges in ensuring that the program is simulated correctly, and challenges in minimizing run-time overheads involved in switching between native and emulated modes of simulation and in evaluating macromodels. Ensuring functional correctness is a challenge that arises at run-time due to the interaction between simulated and natively executed code. On the other hand, overheads involved in hybrid simulation are determined to a large extent by the "mix" of functions simulated in native and emulated modes. We demonstrate in Section 3.3 that these overheads can be predicted statistically and bounded at compile time, *i.e.*, before the hybrid simulation begins.

In the remainder of this section, we illustrate challenges related to ensuring functional correctness.

Natively simulated functions (native functions for short) operate in a different execution environment from functions executed in the simulator (emulated functions). The execution environment includes the processor and memory state, and operating system (OS) services and data structures (such as open file descriptors, sockets, *etc.*). Native functions execute directly on the simulation host processor, and use the host memory space (as part of the simulator). Emulated functions, on the other hand, see models of the processor state and memory that are maintained by the ISS. Run-time interaction of native and simulated functions leads to two distinct challenges:

```
1     T gvar;
2     T *gpointer;
3     main (){
4             T a[10]
5             f1 (a, 10);
6     }
7     f1(T a[], int n){
8             T d, b[8];
9             gpointer = (T*)malloc (n * sizeof(T));
10            gvar = a[i];
11            gvar = b[i];
12            if (n > 1)
13                    gpointer = b;
14            else
15                    gpointer =a;
16            d = **gpointer;      }
```

Figure 1: Example showing challenges in hybrid simulation

Control/data transfer between simulated and native functions: In our hybrid simulation methodology, control transfer between native and simulated functions is restricted to occur at function call boundaries. Thus, a function that is simulated in the ISS may call a native function, or vice-versa. Function calls also involve data transfer in the form of function arguments and return values. Function arguments and return values might be complex data structures that use composite types (*e.g.*, `union/struct/array` in C), or may employ pointers. For real-world applications, a hybrid simulation methodology needs to support automatic transfer of complex data types. For example, consider the simple program shown in Figure 1. Suppose that we want to natively execute function `f1`, while simulating the rest of the program in the ISS. The code shown in Figure 1 is cross-compiled and the target architecture binary made available to the hybrid simulator. In addition, the code for function `f1` is compiled to the host computer, and linked with the simulator for native execution. When function `main` calls `f1`, the ISS must recognize that it needs to transfer control to the native version of `f1`, and pass to it the appropriate values for arguments `a` and `n`. Since argument `a` is an array, and may be passed as a pointer, it is necessary to ensure that the natively executed code is able to understand and de-reference the pointer to access elements of `a`.

Memory synchronization: Native and emulated functions can also interact indirectly by accessing/modifying shared program variables, or by making OS calls that can affect the program environment. Program correctness requires that any modifications to the environment be visible to both native and emulated functions. As an example of indirect interaction, let us again consider the native execution of function `f1` shown in Figure 1. Consider line 10 in function `f1`, which assigns array element `a[i]` to variable `gvar`. This simple operation illustrates the following challenges:

1) Global variable `gvar` is not available directly to the native implementation of `f1`. Any access to a global variable during program execution is accomplished using a virtual address assigned statically by the compiler, or dynamically by the loader if the variable is declared in a shared library. In our example, `f1` has not been compiled against code that declares `gvar`. Therefore, `gvar` must be made available to `f1` by special instrumentation at compile time. Moreover, the virtual address of `gvar` is not a valid virtual address in the address space of the simulation host. It is actually an *index into the virtual memory model* maintained by the simulator, and must be explicitly used as such during the native execution of `f1`. An ISS typically provides a function to translate target virtual addresses into host addresses. These translated addresses might then be treated as normal pointer variables in the native execution domain, and used to access ISS-modeled target memory. We refer to the problem of managing accesses from two different virtual address spaces as *memory synchronization.*

2) Memory synchronization is also a challenge when pointers are used. In function `f1` of Figure 1, the array access `a[i]` (or equivalently the pointer access `*(a+i)`) accesses memory at a simulated virtual address. Therefore, pointer `*(a+i)` must be explicitly translated before it is de-referenced. However, pointers pose another problem that is not encountered with global variables. Some pointers may be local to the native execution domain, *e.g.*, local variables in native functions. For example, pointer `b` used in line 11 of Figure 1 does not point to a simulated virtual address, and explicit translation should not be performed. Thus, with pointers, it is necessary to distinguish between host and simulated target addresses at compile time. This is a variant of the well-studied problem of *pointer analysis* [11]. Complete resolution of pointer targets at compile-time may be impossible, but, in our context, this only limits the set of candidate functions that can be considered for native execution.

3) An important case of the memory synchronization problem occurs due to the use of dynamic memory allocation. Consider the call to `malloc` in function `f1` of Figure 1. This call, if executed directly in the native domain, will allocate memory space in the simulation host processor's address space, causing memory de-synchronization when the simulation returns from `f1` - pointer `gpointer` now points to a memory location in the host address space, and cannot be accessed in the emulated domain. One solution to this problem is to force `malloc` to always trigger a switch to the ISS domain, at the cost of limiting the speed-up achieved due to native execution. A better solution, which we adopt, is to ensure that `malloc` calls from native functions are executed natively, but co-ordinate with the emulated `malloc` to work in the target memory space.

3. METHODOLOGY

Figure 2 presents an overview of the proposed hybrid simulation methodology. The methodology consists of two phases - a characterization phase, which involves constructing macromodels and is performed only once for a given program, and a simulation phase, which consists of native function selection, code instrumentation, compilation, and execution, and is executed for each simulation run. The focus of this paper is on the second phase.

Macromodels are constructed using the methodology presented in [12]. Given the function source code, and a set of training input instances, the methodology automatically generates the macromodel along with mean and variance statistics for expected model error.

The second phase of the methodology consists of profiling, instrumentation, and compilation in order to select native functions and integrate them with the ISS. The target application is profiled on the simulation host computer to obtain execution time statistics and a dynamic function call graph. These are then used in conjunction with macromodel error statistics to guide optimum selection of functions for native execution. Based on this selection, two different versions of the program are automatically generated for cross-compilation and native compilation, respectively. In the cross-compiled version of the program, native functions are replaced in the source code by automatically-generated place-holders called *stubs*. Stubs encapsulate the control and data transfer necessary to invoke a native function. The natively compiled version of the program consists of *drop-in replacement functions* (or simply drop-ins) for each function that was selected for native execution. A drop-in is functionally equivalent to the function from which it is derived, but contains additional code that ensures memory synchronization when it is executed natively. Drop-ins and energy macromodels for native functions are compiled and statically linked with the executable of the ISS itself to obtain the hybrid simulator.

Figure 2 depicts as rectangles the three main tools involved in the second phase, *viz. Stubgen*, *Native Drop-in Generator* and *Native Function Selector*. Next, we examine each of these tools in detail, and discuss their roles in enabling hybrid simulation.

3.1 Stubgen

Calling native functions from emulated code is accomplished using a mechanism similar to remote procedure calls. The tool *stubgen*

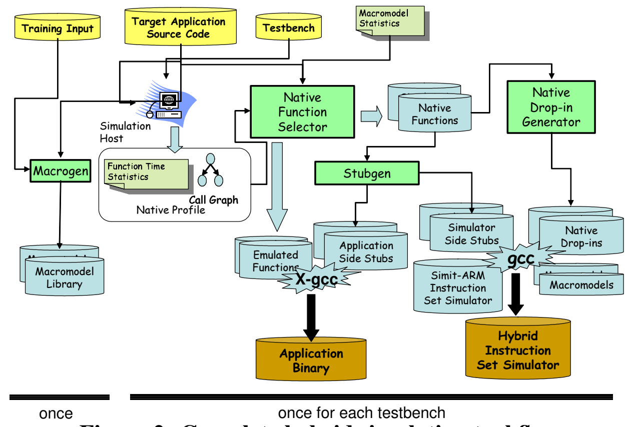

Figure 2: Complete hybrid simulation tool flow

parses application source code to generate two stubs for each native function, an application side stub and a simulator side stub. It is the job of the stubs to transfer function arguments and return values between native and emulated functions. The application side stub is a place-holder function that is compiled with the target application and is used by the simulator at run-time and at initialization.

At run-time, the application side stub simply transfers control to the simulator side stub whenever it is called. Control transfer is accomplished using a special system call that we created for this purpose, called `switchNative`. `switchNative` expects one argument, a number which enables system-call handling code in the simulator to identify the correct simulator side stub to invoke.

The application side stub of a native function exports the same interface as the corresponding native function. This enables the cross-compiler to insert debugging information identifying the register/memory locations used to pass function arguments. This information, present in the application binary, is used by the hybrid simulator to discover function argument addresses when the program is first loaded into the simulator.

The simulator side stub is code that is complied with the ISS, and uses argument addresses discovered at load time to pass arguments to the native drop-in. The simulator side stub also writes back the value returned by the drop-in function so that it is available to the calling function once control has been returned to the emulation domain. Through the mechanisms described above, stubgen enables transparent control and data transfer between emulated and native functions.

3.2 Native Drop-in Generator

Application functions cannot be simulated directly in native mode due to the problems of memory de-synchronization and unknown pointer targets. The native drop-in generator instruments application functions to make them suitable for native simulation.

The drop-in generator performs two tasks. We illustrate these using the function `f1_dropin` in Figure 3 that was generated by the drop-in generator from function `f1` in Figure 1. It is worth noting that every drop-in is passed a reference to an ISS [13] object, which is a C++ data structure that encapsulates all the simulator functions that the drop-in needs to call.

Resolving global variables: The drop-in generator detects use of global variables in the analyzed function, such as `gvar` in Figure 1. As we saw earlier, native functions are not aware of such global variables since they have not been linked against code containing them. For each such global variable of type `T`, found in any native function, the drop-in generator inserts a declaration of a pointer of type `extern T*` in the simulator side stub. The ISS uses debugging information at program load time to initialize these global variable pointers to proper addresses. Since these pointers refer to the simulated memory space, the drop-in generator also needs to insert code to explicitly translate them prior to de-references, as described next.

Pointer analysis: A pointer that points to a location in simulated memory cannot be de-referenced directly. Instead, as Figure 3 shows, the drop-in utilizes the macro `TRANSLATE`, which can use pointer values to access the simulated memory model maintained by the ISS. Simulated memory is maintained as a data structure in *host* memory and the value returned by ISS function `TRANSLATE` is a host memory pointer that is equivalent to a location in the simulated memory. The translated pointer may be typecast and de-referenced directly by native code. Figure 3 illustrates this process with a few examples.

```
 1  extern T* gvar;    /* resolved at program load by ISS */
 2  extern T** gpointer;
 3  #define TRANSLATE(x,t)
 4    reinterpret_cast <t> (emu->mem->translate (x))
 5  f1_dropin (T a [], int n, emulator *emu){
 6    T d, b[8]; /* local array declaration*/
 7    *(TRANSLATE (gpointer, T**))
                              = (T*)Hysim_malloc (n*sizeof (T));
 8        *(TRANSLATE (gvar, T*)) = b[1];
 9        *(TRANSLATE (gvar, T*))
                              = *(TRANSLATE (&a[1], T));
10        if (n > 1)
11                *(TRANSLATE (gpointer, T**)) =  b;
12        else
13                *(TRANSLATE (gpointer, T**))
                              =  (TRANSLATE (a, T*));
14      d = **gpointer; /* **gpointer cannot
                              be resolved at compile time */ }
```

Figure 3: Example drop-in function

Note that array access $b[1]$ in Figure 3 does not use the translation step, since it is a local variable in the native function. The drop-in generator needs to distinguish at compile time between pointers that require explicit translation and those which do not, based on the possible memory locations that the pointer may point to.

The set of memory locations pointed to by a C pointer or array variable, called its *range*, may be represented abstractly [11] using an integer interval domain and a set *Loc* of abstract representatives of locations, or objects defined in the program to which a pointer may legally point.

Definition 1 *Loc* = $\{v \mid v$ *is a variable in the program*$\} \cup \{\texttt{malloc}_i \mid i$ *is a program point where* `malloc` *is called*$\}$.

As an example, variable $b[1]$ is associated with integer domain $\{1\}$ and has $Loc_{\texttt{b[1]}} = \{*b\}$. Range $Loc_{\texttt{**gpointer}}$ for the variable assigned in line 13 of Figure 3 is $\{*a, *b\}$. This abstract representation can be used to state the following test to determine the need for address translation.

Theorem 1 *Define for native function f the set $NATIVE_f = LOCAL_f \cup NATIVE\text{-}ARG_f$ where $LOCAL_f$ is the set of all local variables in f and $NATIVE\text{-}ARG_f$ is the set of all function arguments of f that do not need to be translated. Then a pointer variable p in f does not need to be translated if $Loc_p \subset NATIVE_f$. p should be translated if $Loc_p \cap NATIVE_f = \Phi$. If none of these conditions holds, it is not possible to decide at compile time whether p should be translated or not.*

In practice, the ambiguity in resolution of pointer targets implies that not all functions are amenable to native simulation. Such functions are precluded from being selected as native functions in our hybrid simulation methodology.

Finally, as indicated in Section 2, all `malloc` calls, whether called from a native or emulated function, must allocate space from the same area in the simulated memory space. We achieved this by treating the `malloc` function itself as a native function, so that all calls to `malloc` from emulated or native code are executed by a drop-in for `malloc` that is memory-synchronized to the `malloc` executed within the simulator. Thus, all calls to `malloc` allocate space in the *heap area of simulated memory.*

3.3 Native Function Selector

In this section, we present models for estimation error and simulation time that the hybrid simulator can be expected to obtain in simulating a program over a given input. We then discuss our algorithm to optimally choose the set of native functions, in order to minimize simulation time under a probabilistic maximum constraint on the expected energy estimation error.

An execution instance of a program over a given input might be abstractly represented as a dynamic function call graph, constructed based on observing program execution under the given input. We use a dynamic call graph annotated with execution/simulation time statistics, and characteristics of function macromodels to obtain estimates of simulation error and simulation time, which are used to guide the optimal selection of natively executed functions.

Bounding simulation error: We seek to provide to the user a probabilistic bound on estimation error ϵ of the form,

$$\text{Probability}(|\epsilon| \leq \epsilon_o) \geq \alpha_o, \quad (1)$$

where ϵ_o and α_o are user specified tolerance and confidence values.

Characterization based macromodeling techniques [9, 12], used in this work to construct macromodels, are based on the normality assumption, employed widely in classical regression. This allows us to derive a bound of the form given in Equation (1). The total energy estimate reported by hybrid simulation is a sum of energy estimated using instruction-level models, denoted by E^I, and macromodels. For our purpose, E^I is an exact value. We assume that energy estimates reported by the macromodel for function f_i have normally distributed errors of the form $\epsilon_i = N(0, \sigma_i)$. We claim, without proof, that the total energy E has error $\epsilon = N(0, \sum_{i \in F} N_i\sigma_i)$, where F is the set of all functions in the program and N_i is the number of calls in native mode to function i. It can be shown using properties of the normal distribution that under the constraint given by Equation (1),

$$\sum_{i \in F} N_i\sigma_i < \frac{\epsilon_o}{\text{erf}^{-1}(\frac{\alpha_o}{2})} = \Delta_{max}, \quad (2)$$

where Δ_{max} is the maximum allowable deviation of the expected estimation error. We have thus derived a constraint on the total standard deviation of macro-model errors, which may be used to guide mode-selection in order to minimize expected simulation time.

Minimizing simulation time under an error constraint: The time required by the hybrid simulator to complete simulation on a given input may be bounded using time estimates, obtained while profiling, for emulated and natively executed functions, as follows.

$$T^H_{main} = \sum_{f \in F} \sum_{j=1}^{N_f} T^N_f(j) + T^I, \quad (3)$$

where T^I is the time required to simulate all emulated functions and $T^N_f(j)$ is the time spent in simulating call j to function f in the native mode. We can thus formulate the problem of minimizing total simulation time as a minimization problem in variables N_i with the objective specified by Equation (3) under the constraint given by Equation (2). Thus, the problem is to minimize the expected simulation time by deciding before simulating each call to a function, the mode it should be simulated in.

As stated above, the problem requires a *run-time* solution, since it is not possible to enumerate all possible calls to a function at compile time (*i.e.*, before simulation). The problem may be solved at compile time by imposing the restriction that all calls to a function from a particular call site be simulated in the same mode. We use a dynamic programming based algorithm to solve the mode-selection problem. Our algorithm is pseudo-polynomial in the maximum allowable error deviation Δ_{max}. However the complexity does not seem to be an issue in practice, as algorithm runtime on each of our experiments was less than a minute - much less than the simulation times.

4. IMPLEMENTATION AND RESULTS

In this section, we discuss implementation details and present the results of our experiments with hybrid simulation. The tool flow presented in Figure 2 is based around a C parser and a number of `Python` scripts to glue together various tools. The hybrid simulator itself runs on an x86/Linux platform and is based on the source code of the `SimIt-ARM` emulator `ema-count` [13]. It simulates a little endian target processor, and can execute programs cross-compiled using the ARM-Linux-gcc toolchain. Simulated ARM binaries must be statically linked and contain debugging information about variable and function argument locations.

We evaluated our hybrid simulation technique using standard embedded benchmarks from MediaBench [14], MiBench [15] and a complex biometric application called the Fingerprint Verification System (FVS) [16]. FVS was chosen because of its challenging workload - almost an entire work week is required to simulate the authentication of a fingerprint image on the baseline ISS. For MediaBench benchmarks, the standard supplied data set was used for training macromodels, and the alternate data set, also available as part of the suite, was used for benchmarking the simulator. Fingerprint images available from [16] were used as inputs for FVS. Descriptions of GSM toast, GSM untoast, mpeg2dec and mpeg2enc may be found in [14]. SHA is the SHA encryption algorithm, taken from [15].

Table 1: Example Applications

Benchmark	Time (secs)		Speed-up		Error
	Baseline	Hybrid	Expected	Observed	
FVS	120 hrs	40 hrs	68%	66%	1%
GSM toast	1135	516	59%	55%	2%
GSM untoast	11.5	4.5	76%	62%	4%
MSG mpeg2dec	1116	9	99%	99%	1%
MSG mpeg2enc	356	143	77%	60%	6%
SHA	17	1	99%	94%	1%

For hybrid simulation, native functions were selected using a requirement of 5% or smaller error, with probability 95% or more. We compared simulation time and energy estimates with `SimIt-ARM`, enhanced with instruction-level energy models from Jouletrack [17], as the baseline case. The results of our experiments are presented in Table 1. For each benchmark, we report the time required to complete simulation using the baseline ISS as well as the hybrid simulator. We also compare the speed-up observed in simulation time with the expected value, *i.e.*, the value predicted by the native function selection algorithm. Finally, the error observed in energy estimates is also reported. The times consumed by the native function selector, native drop-in generator and stubgen are not reported since they were insignificant compared to simulation times. The average speed-up obtained was 70%.

5. CONCLUSIONS

In this work, we presented a function-level hybrid simulator that can be used at an early stage in software design to obtain high fidelity energy estimates with significant speed-up over pure instruction set simulation. Our methodology does not rely on application characteristics and greatly extends the applicability and usability of macromodel based energy estimation.

6. REFERENCES

[1] P. Magarshack and P. G. Paulin, "System-on-chip beyond the nanometer wall," in *Proc. Design Automation Conf.*, June 2003, pp. 419–424.

[2] J. Zhu and D. D. Gajski, "A retargetable, ultra-fast instruction set simulator," in *Proc. Design Automation & Test Europe Conf.*, Mar. 1999, p. 62.

[3] 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. Design Automation Conf.*, June 2002, pp. 22–27.

[4] W. S. Mong and J. Zhu, "Dynamosim: A trace-based dynamically compiled instruction set simulator," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2004, pp. 131–136.

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

[6] W. Qin and S. Malik, "Flexible and formal modeling of microprocessors with application to retargetable simulation," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2003, pp. 556–561.

[7] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proc. Int. Conf. Parallel Architectures & Compilation Techniques*, 2003.

[8] 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.*, 2003, pp. 504–509.

[9] 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.

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

[11] S. H. Yong and S. Horwitz, "Pointer-range analysis," in *Proc. Int. Static Analysis Symp.*, Aug. 2004, pp. 133–148.

[12] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha, "Automated energy/performance macromodeling of embedded software," in *Proc. Design Automation Conf.*, June 2004, pp. 99–102.

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

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. Int. Symp. Microarchitecture*, Nov. 1997, pp. 330–335.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. Wkshp. Workload Characterization*, Dec. 2001, pp. 3–14.

[16] S. Patel, "Finger-print verification system." [Online]. Available: http://fvs.sourceforge.net/

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