Abstract System-Level Models for Early Performance and Power Exploration

Andreas Gerstlauer, Suhas Chakravarty, Manan Kathuria, and Parisa Razaghi
Electrical and Computer Engineering
The University of Texas at Austin
gerstl@ece.utexas.edu, suhas.chakravarty@utexas.edu, manan@austin.utexas.edu, parisa.r@mail.utexas.edu

***Abstract*—With increasing complexity of today's embedded systems, research has focused on developing fast, yet accurate high-level and executable models of complete platforms. These models address the need for hardware/software co-simulation of the entire system at early stages of the design. Traditional models tend to be either slow or inaccurate. In this paper, we present ingredients for a class of abstract, high-level platform models that enable fast yet accurate performance and power simulation of application execution on heterogeneous multi-core/-processor architectures. Models are based on host-compiled simulation of the application code, which is instrumented with timing and power information. Back-annotated source code is further augmented with abstract OS and processor models that are integrated into standard co-simulation backplanes. The efficiency of the modeling platform has been evaluated by applying an industrial-strength benchmark, demonstrating the feasibility and benefits of such models for rapid, early exploration of the power, performance and cost design space. Results show that an accurate Pareto set of solutions can be obtained in a fraction of the time needed with traditional simulation and modeling approaches.**

I. Introduction

In recent years, rising hardware and software complexity in embedded systems has necessitated the elevation of the design process to a higher level of abstraction. At the system level, executable models of entire platforms can be built that enable hardware-software co-development and rapid, early design space exploration. Such models provide quick feedback to the designers about the effect of their design decisions on critical system metrics like performance, power consumption, and system cost. Complex interactions and the highly dynamic nature of systems make their static analysis difficult, which is why such executable models are indispensable.

An overview of different platform modeling approaches is shown in Fig. 1, organized by granularity of computation and communication. Models of Computation (MoCs), such as process networks, dataflow models or state machines are at the most abstract level, describing application execution as purely functional tasks that exchange messages over channels for communication and synchronization. At the other end of the spectrum are RTL models, which contain micro-architectural details and are cycle-accurate but slow. In between, various system modeling approaches have been developed to trade-off simulation speed and accuracy. On the communication side, transaction-level modeling (TLM) concepts abstract away from pins and cycles to provide faster simulations at good accuracy. This includes network simulators, which describe communication over wired or wireless media at the level of packet transactions. On the computation side, instruction-set simulators (ISSs) are used to emulate execution of an instruction stream on an abstracted model of the target processor in either micro-architectural or purely functional form. When combined with TLM approaches, cycle-accurate or virtual platform simulations of complete systems can be constructed, respectively. These, along with RTL/gate level models, have traditionally been used to perform performance and power simulations. Their drawback is that, depending on the level of detail, they are either inaccurate or slow. This is especially true in large, full-system multi-core/-processor contexts.

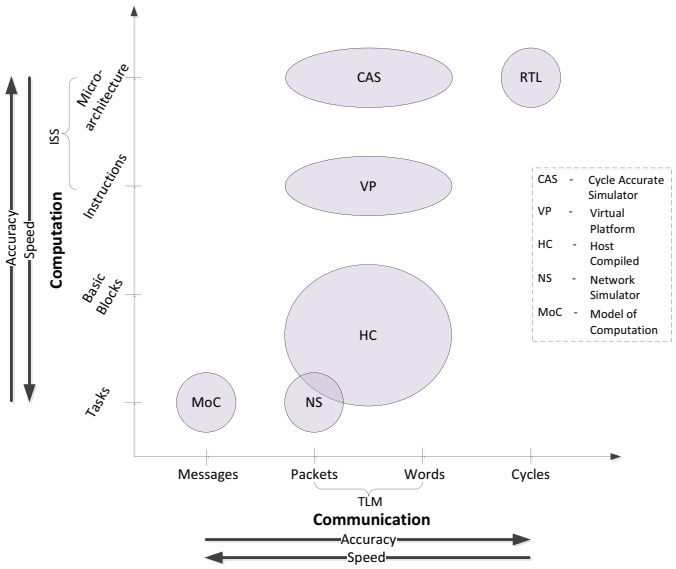

Fig. 1. Modeling space.

As an alternative to ISS-based models, high-level software and processor models based on native, host-compiled software execution have recently become popular [1]. The underlying concept is to follow TLM ideas from communication and push computation modeling to higher abstractions above the level of instructions [2]. Host-compiled approaches describe computation at the source code level (typically in C-based form). This allows a functional model to be natively compiled onto the host for fastest possible execution. Timing and other information, such as energy consumption, are added through

This research was partially supported by SRC Task 2085.001.

978-1-4673-0772-7/12/$31.00 ©2012 IEEE

back-annotation of the source code at a particular granularity. That granularity should be below complete functions or tasks, such that dynamic effects of program execution paths can be simulated, but simulation speeds remain fast. Lastly, the back-annotated application is wrapped into lightweight models of operating systems and processors that plug into standard TLM backplanes to accurately simulate the impact of the execution platform. Host-compiled models seek to provide the best tradeoff between speed and accuracy. As such, they complement existing modeling solutions specifically to enable rapid yet precise early design space exploration.

In this paper, we present a comprehensive host-compiled modeling approach that aims to incorporate both power and performance metrics for fully heterogeneous multi-core and multi-processor platforms. Previous host-compiled approaches (as summarized in Section II) have thus far only focused on timing (performance) simulations for limited platforms, e.g. restricted to single-core processors. By contrast, we propose a model that is built in a flexible and retargetable way by annotating the application code, at the basic block level, with both timing and energy consumption estimates (see Section III). Back-annotated application code is further augmented with fast and accurate host-compiled multi-core RTOS and processor models, as described in Section IV. Finally, in Section V, we apply our approach to a representative, industrial-strength example in order to qualitatively and quantitatively demonstrate the ability for rapid exploration of the design space across power, performance and cost objectives. Results show that a Pareto-optimal front of architecture candidates can be determined in a fraction of the time taken using traditional ISS-based virtual platform capabilities.

II. Related Work

A range of so-called host-compiled or source-level modeling concepts have been researched in recent years. Some of the earliest approaches were centered around models of the OS itself [3]–[5]. Later, these were extended into complete processor models that include timing-accurate descriptions of interrupt chains and TLM-based bus interfaces [6], [7]. Such processor models have been shown to simulate at speeds beyond 500 MIPS with more than 95% timing accuracy.

Application source code is usually back-annotated at a basic block level [2], [8] to accurately capture dynamic data dependencies while minimizing overhead. Several nearly identical approaches [9]–[11] use a standard compiler frontend to first bring the code down to an intermediate representation. This allows typical source-level compiler optimizations to be accurately considered. Another approach is to use intelligent algorithms to map binary code blocks and associated timing characterization directly to the source level [12], [13].

In all cases, back-annotation is based on static emulation of source code execution on a timing model of the target processor. In some cases, this may be as complex as using the target tool-chain to compile and simulate each block on a cycle-accurate target model. Within such a framework, most existing approaches are tied to a specific backend target.

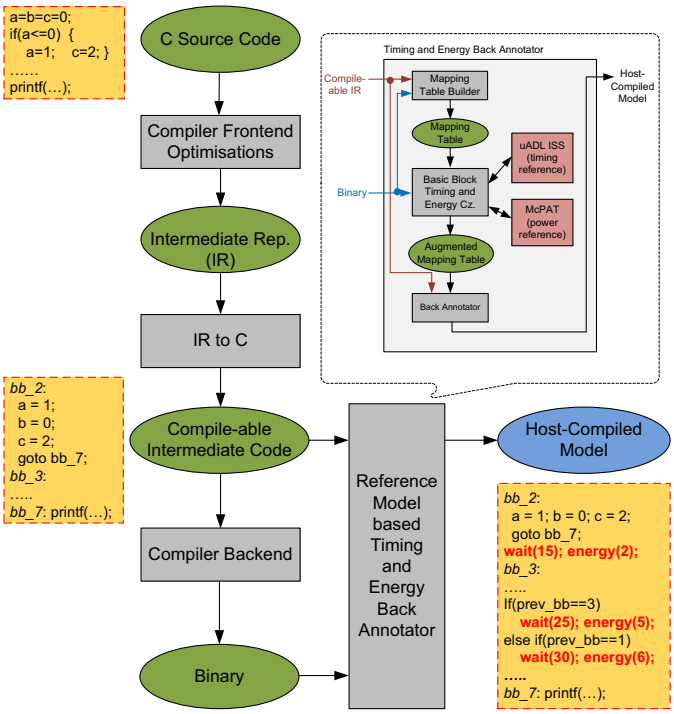

Fig. 2. Back-annotation flow.

Several host-compiled approaches also support hybrid static and dynamic timing models and back-annotation, either by including dedicated simulation models of critical dynamic micro-architecture features, such as caches or branch predictors [8], [9], [14], or by toggling between host-compiled and ISS-based models dynamically at simulation time [15], [16].

For power estimation, popular approaches are to develop macro models for micro-architectural functional blocks [17], [18] or use analytical modeling frameworks [19], [20]. These utilize activity information gathered from cycle accurate performance simulators to estimate power consumption. Tiwari *et al.* [21] present an instruction level power model. However, their model is not portable in that it requires detailed profiling of the instruction set of the target. While our work leverages existing low-level power reference models, to the best of our knowledge, no true host-compiled power estimation work at the level of basic blocks exists.

III. Application Modeling

In this section, we describe our methodology for obtaining an abstract model of the application back-annotated with timing and power metrics for execution on a given target. This model does not take into account the remaining platform environment, such as interactions with an RTOS or processor. Rather it seeks to provide a basis for building complete models of a whole platform as described in later sections.

A. Back-Annotation Flow

Fig. 2 shows the proposed flow for developing host-compiled models, accompanied by representative code snippets at various stages. The application C code is passed through a generic cross-compiler front end (we use the gcc

compiler suite for our approach), which performs optimizations and produces an intermediate representation (IR). Working with the IR allows us to take into account all typical front-end optimization passes. Actual estimation and back-annotation hence works at the IR level. This IR is massaged (via IR to C conversion) into a compileable form, which is then passed to the generic cross-compiler backend for generation of the binary. The assumption is thereby that the backend does not introduce any major changes in the control flow graph (CFG) of the application and hence it is sufficient to work at the IR level to capture all data-dependent execution behavior.

The next step, illustrated in the inset in Fig. 2, is to determine the mapping between basic blocks in the compileable IR and addresses in the binary. The resulting mapping table is needed for extracting individual basic blocks from the binary. This process can be automated using debug information that associates a certain line number in the source code with the address in the binary of the first assembly instruction that corresponds to the source code line. To be able to use debugging information for this purpose, the binary has to be compiled from the IR, since the mapping for back annotation is desired to the compileable IR and not the original application code. The fact that the compiler backend does not introduce any major changes in the CFG allows us to find accurate mappings. However, in a few cases we observed that backend control-flow optimizations were present in the binary generated from the IR. A constraint is therefore introduced that the generation of the binary from the compileable IR should be done with such optimizations turned off. This does not significantly affect the final accuracy of the model, but is necessary to get rid of artifacts of the two-step compilation process.

In the next step in the flow, each basic block extracted from the binary is then characterized by executing it on a retargetable cycle-accurate ISS that is complemented with a reference power model, such as McPAT [20]. The timing and energy consumption of a basic block depends on the state of the processor at the start of execution of the block. In a real execution flow, this state is determined by code that has already been executed. To approximate this effect during characterization, pair-wise executions of a particular basic block and its possible predecessors are carried out. For cycle-accurate simulation of execution time on the target, we use a retargetable ISS that is based on an open-source micro-architecture description language (uADL) [22]. For each characterization run for a single basic block, the final state of registers and data memory locations for a particular preceding block is used as the processor initial state. For power estimation, execution statistics, such as number and types of instructions executed, the number of times different functional units and memory are accessed, cache misses and hits etc. are extracted by analyzing the trace of the ISS execution of a given basic block. McPAT also requires the configuration of the target processors, which is provided separately. The per-block power consumption reported by McPAT is then converted into an energy consumption metric, utilizing the knowledge of the execution time for that block.

TABLE I
APPLICATION MODEL ACCURACY.

	Host-Compiled	**ISS/McPAT**	Error
Timing [cycles]	22,864,740	22,864,730	0.0004%
Energy [mJ]	222.6	241.4	7.8%
Sim. Speed [MIPS]	2000	0.8	-

Thereafter, the mapping table is augmented with timing and energy consumption numbers and is used for directing the annotation of the compileable IR at the correct points. For reasons explained earlier, a function is annotated, which returns the appropriate timing/energy figure depending on the predecessor of the current basic block. This results in the back-annotated, host-compiled application model, which is then natively compiled for the simulation host and executed to obtain overall performance and energy consumption figures for the application running on the given target processor. An average power consumption profile can be computed by dividing the total energy consumed by the total time taken, as reported by the host-compiled model.

B. Back Annotation Experiments and Results

To demonstrate the feasibility and benefits of our approach, we applied it to a generic PowerPC dual-issue core with a static branch predictor and no cache, MMU, or Floating Point Unit (FPU). A custom application implementing "Eratosthenes' Sieve" was used to develop and validate the timing and power (energy) back annotation flow. The application finds prime numbers in a given range starting from 0. For our experiments, the range was 500000. Table I shows the timing and energy numbers of the host-compiled simulation as compared to execution on the reference model. Result show a negligible timing error of 10 cycles in the host-compiled model compared to the cycle-accurate ISS. The error in the energy consumption calculated by the host-compiled model is about 8% when compared against the results obtained from the ISS + McPAT combination. As future work, we plan to investigate the sources of this error. Finally, the comparison of simulation speeds shows a big speedup of 2400x for the host-compiled model compared to the cycle-accurate ISS execution.

IV. PLATFORM MODELING

In the previous section, we described source-level modeling of applications into which power consumption and execution delays corresponding to a target processor are back-annotated. In order to evaluate the real-time performance and power consumption of a complete platform, we need to further embed applications into accurate models of their complete execution environment. For this purpose, we introduce abstract RTOS and processor models that can be integrated through standard TLM backplanes. Fig. 3 outlines the structure of the proposed platform model. At the highest level, the user application consists of a set of back-annotated high-level tasks that are controlled by and interact with an underlying OS model. The OS model manages the scheduling of application tasks across available cores. A hardware abstraction layer (HAL) includes models of necessary I/O drivers and implements an

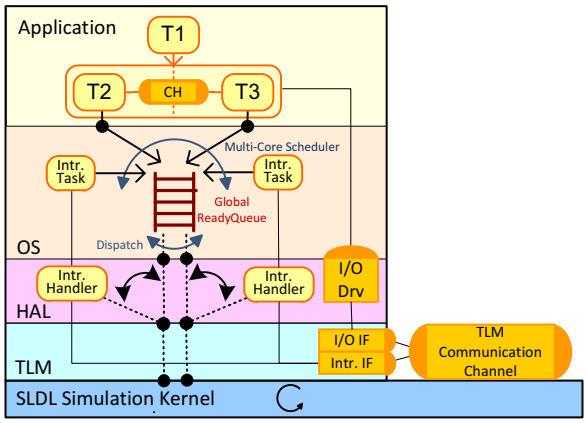

Fig. 3. Host-compiled, multi-core platform model.

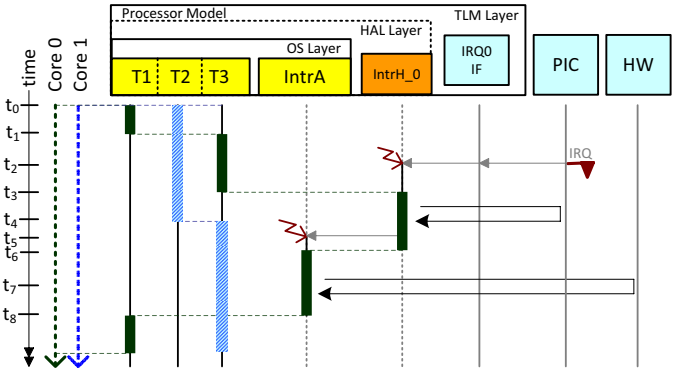

Fig. 4. Processor modeling trace.

abstract interrupt handling mechanism. The HAL combined with a TLM layer provide a high-level processor model that interfaces with the TLM backplane, which provides a fast system-wide co-simulation. The complete platform model is developed on top of a standard system-level design language (SLDL) simulation kernel, which provides basic concurrency and discrete event handling on a host machine.

A. OS Modeling

An abstract RTOS model replicates a typical multi-core OS kernel to emulate the execution of high-level tasks across available cores [23]. The OS model wraps around the basic SLDL event handling mechanism and ensures that at any time only as many SLDL threads as there are cores are active. Our OS model provides the facilities to integrate an application with OS services through a canonical API that supports OS initialization and startup, task management, execution delay modeling, and event synchronization.

The OS model internally consists of typical queues that maintain the state of tasks running on the processor. Tasks are transfered between these queues whenever API methods are called: a *Ready* queue holds tasks that are ready to execute and is sorted based on a user-defined scheduling policy. An *Idle* queue holds periodic tasks that have called the kernel's method to finish their current job execution and sleep until the start of the next period. Tasks waiting for an event are transfered to a *Wait* queue and are placed back in the *Ready* queue when a method is called to release them. Finally, a *Sleep* queue holds tasks that have been suspended until an active task calls a method to resume them.

At the core of the OS model is the multi-core scheduler. The scheduler is an internal function of the OS model and is called by the OS API methods whenever a task switching is possible or required. The main functionality of the scheduler is to retire the currently active task on a core, if any, and place it in a proper queue and assign a new task from the *Ready* queue to that core. The OS kernel supports both partitioned and global scheduling schemes, which are distinguished by the number of *Ready* queues associated with each core. In a partitioned scheme, each core has a separate ready queue and tasks are initially assigned to a fixed core and queue. The OS picks tasks for a core only from the associated queue, but it can perform load balancing to migrate tasks between queues and cores either at regular intervals or whenever a task leaves a core. In the global scheme, the OS maintains only a single *Ready* queue and tasks can be freely assigned to the next available core based on a used-defined cores affinity. In both schemes, the scheduler can organize the *Ready* queues based on different scheduling policies including time-sliced round-robin, FIFO, or static priority. In addition to basic OS services, the OS kernel simulates back-annotated task execution delays using underlying SLDL primitives wrapped inside an OS API `TimeWait()` method. After advancing the simulation time at the SLDL level, this method calls the OS scheduler to allow for preemption of the current task by any higher priority task that became available in the meantime.

Overall, using the presented OS model, the designer can easily adjust the OS kernel for a desired application to meet system requirements. Overall, our OS model helps designers to evaluate the real-time behavior of applications across different scheduling policies and schemes on platforms with different number of cores.

B. Processor Modeling

The OS model is combined with the HAL and TLM layer to form a complete multi-core processor model (Fig. 3). The HAL and TLM layers contain any necessary bus drivers and interfaces to integrate the processor model into a TLM platform. The HAL layer implements an accurate and flexible model of a general multi-core interrupt handling chain. An external programmable interrupt controller (PIC) model manages interrupt sources and generates the interrupt request (IRQ) signals for each core. The TLM layer contains processes that listen for interrupt requests and trigger corresponding interrupt handlers in the HAL. Interrupt handlers are modeled as special tasks associated with each core, created via API methods exported by the OS layer. Interrupt handlers in turn communicate with the PIC and trigger regular interrupt tasks in the OS layer for a specific interrupt source. Finally, user-supplied code in the interrupt tasks can communicate with external hardware, with application tasks or with the OS model, e.g. to spawn additional processing tasks.

Fig. 4 shows an example of a simulated task execution and interrupt handling sequence for an application with three tasks

TABLE II
OS MODEL ACCURACY AND SPEED EVALUATION.

Task Sets	**Small Sets**	**Medium Sets**	**Large Sets**
Avg. Tasks #/Core	11	4	3
Avg. Core Utilization	0.6	0.7	0.7
Avg. Err. (1μs)	0.5%	0.25%	0.11%
Avg. Err. (10μs)	0.71%	0.22%	0.10%
Avg. Err. (100μs)	0.64%	0.69%	0.43%
Avg. Err. (1000μs)	10.3%	6.46%	4.0%
Speed [GIPS] (1μs)	0.15 GIPS	0.13 GIPS	0.13 GIPS
Speed [GIPS] (10μs)	1.68 GIPS	1.3 GIPS	1.17 GIPS
Speed [GIPS] (100μs)	10.5 GIPS	8.2 GIPS	8.0 GIPS
Speed [GIPS] (1000μs)	23 GIPS	31 GIPS	36 GIPS

running on two cores. Tasks indices are ordered by decreasing priorities. At time t_0, the two highest priority tasks are running on the two processor cores. At time t_1, Task *T1* is blocked by an external event, and the OS schedules task *T3* on core 0. At time t_2, an external HW component generates an interrupt that is detected by the PIC. The PIC is programmed to route this interrupt to core 0 and consequently sets the IRQ0 signal. This is detected by the TLM layer, which activates the *IntrH_0* handler and inserts it into the OS interrupt queue on core 0. At the next preemption point offered by the back-annotated timing model in the current task on core 0 (task *T3*), *IntrH_0* is then scheduled (time t_3). *IntrH_0* communicates with the PIC to acknowledge the interrupt source. At time t_4, task *T2* is blocked and accordingly, task *T3* is scheduled on core 1. At time t_5, *IntrH_0* releases a corresponding user-level, high-priority interrupt task *IntA* in the OS. Finally, at time t_6, *IntrH_0* completes execution and removes itself from the OS interrupt queue. This results in *IntrA* being scheduled on core 0, which in turn then communicates with the external HW component (time t_7). At the end of sequence (time t_8), the processor resumes execution of normal application task *T1* on core 0. Overall, the combined host-compiled model is able to faithfully replicate the execution sequence of the real platform. However, accurate modeling of task preemptions (e.g. by interrupts) is a function of the back-annotated timing granularity. As such, there exist fundamental tradeoffs between simulation speed and modeling accuracy.

C. Accuracy and Speed Evaluation

To evaluate the simulation performance and accuracy of the proposed OS and processor models, we compared the response time of randomly generated periodic task sets running on our host-compiled model to a virtual reference model of a dual-core MIPS34Kc Malta platform running real binaries on top of a 2.6.24 Linux SMP kernel [24]. Task periods are uniformly distributed over [1, 100] ms, while task utilization are distributed over [0.001, 0.1], [0.1, 0.4], and [0.001, 0.4] for a small, large or medium range of execution delays. In each case, we generated task sets with different numbers of tasks to achieve a variety of light, medium and heavy core loads. Tasks priorities are assigned inversely to their periods following a rate-monotonic scheduling policy. Actual task execution delays were measured on the reference simulator and back-annotated at different levels of timing granularity.

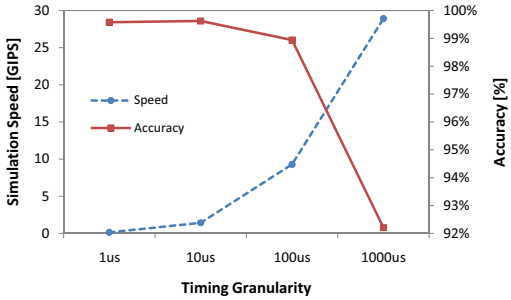

Fig. 5. OS model accuracy-speed tradeoffs.

Table II summarizes the simulation speed and accuracy for different task sets. Model error was measured as the average absolute difference in individual task response times over all tasks and task iterations. Results show that with a timing granularity of 1μs, the average timing error across all task sets is less than 0.5%. The timing error is higher for task sets with a large number of small tasks. This is because we have ignored OS context switch delays in this model. We ran each task set for 10s of simulated time, which at a nominal rate of 100MIPS simulated by the reference ISS corresponds to 1000 million instructions on each core. The reference platform simulates each set for about 30s of wall time. By contrast, our model simulates the same setup in faster than real time with a throughput of more than 1000MIPS per core at timing granularities of 10μs and above.

Fig. 5 plots the tradeoff between accuracy and simulation speed over all task sets. As can be seen, decreasing the timing granularity results in a higher accuracy but comes at a loss in simulation performance. Overall, designers can achieve fast simulation speed while having accurate results by selecting proper timing granularities that fall into an intermediate range. In addition, techniques for automatic timing granularity adjustment can be integrated into the OS model in order to significantly improve the speed-accuracy tradeoff [25].

V. EXPLORATION

To demonstrate the capabilities enabled by our approach, we consider design space exploration for a task set composed out of a modified subset of applications from the automotive category of the MiBench suite [26]. The characteristics of these tasks as determined by back-annotation for a single-core, 100MHz PowerPC target processor are summarized in Table III. The tasks *basicmath_large*, *qsort_large* and *susan_edge+susan_corner* are scheduled to run periodically with periods of 2.5, 2 and 1 second(s) respectively. The output of *susan_edge* is thereby communicated to the input of *susan_corner*, forming a pair of dependent tasks. The resulting task set is run for 10 seconds of simulated time.

TABLE III
MIBENCH TASK CHARACTERISTICS

	Execution time [ms]	Energy [mJ]
Basicmath_large	80.08	83.9
Qsort_large	509.83	541.9
Susan_edge	78.89	83.3
Susan_corner	43.38	46.6

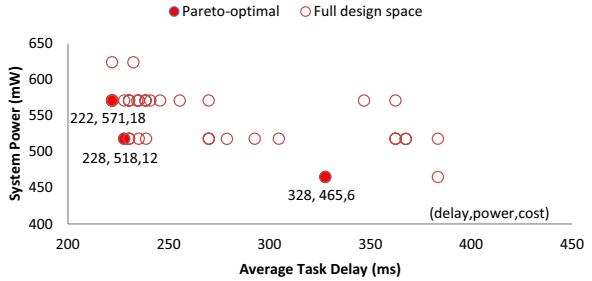

Fig. 6. Design space exploration.

We explored a design space with up to four processors, a choice of two OS schedulers (priority-based or round-robin) and required communication architectures in the form of busses and bridges. This forms a design space of almost 2000 unique architectures. By scripting the SCE [27] framework, we automatically generated host-compiled models with back-annotated delay and energy numbers for all architectures. Simulating those models provides total delay, power consumption and cost for each design alternative. This results in 35 distinct design points (Fig. 6), where the set of Pareto-optimal designs comprises less than 1% of the total design space.

As shown in Table IV, a design space exploration (DSE) approach utilizing host-compiled models is able to evaluate all 2000 design alternatives in 47 hours of CPU time on a cluster of 2.5GHz Intel Xeon workstations. By contrast, we estimate that an ISS-based approach for full design space exploration would take more than 500 hours. This is prohibitive already for such a simple example, requiring compromises in which certain design decisions are chosen manually, which can potentially lead to optimal design alternatives being missed.

Exploration time can be further reduce by exploiting the fact that host-compiled models are constructed in layers. In a two-step exploration, the design space is initially pruned of infeasible designs using variants of host-compiled models that only consider computational effects, i.e. that lack HAL and TLM layers and hence trade off increased simulation speed for an inaccurate (zero-delay) communication model. Only designs close to the Pareto front are then further explored using full host-compiled platform simulations. This allows us to further reduce exploration time down to 8 hours.

VI. Summary and Conclusions

In this paper, we presented a comprehensive approach for host-compiled power and performance modeling of heterogeneous multi-processor and multi-core platforms. Retargetable back-annotation of application code coupled with abstract OS, processor and platform models allows for flexible, fast yet accurate full-system simulations. Experimental results show that host-compiled modeling provides a feasible platform for large design space exploration under high accuracy and fast simulation speed. Generally, different stages of the design process call for models at different levels of abstraction, where host-compiled models at varying levels can play a key role for early and intermediate exploration steps.

Ongoing and future work will require extensions to further increase accuracy and applicability of such models, e.g. by taking into account the data (input) dependent nature of power consumption or by considering more advanced target platform architectures that include caches, out-of-order executions or dynamic branch prediction.

TABLE IV
Design Space Exploration Time

	CPU Time [hours]
System-level full DSE	47
System-level step-wise DSE	8

References

[1] A. Gerstlauer, Host-compiled simulation of multi-core platforms, *RSP*, Jun. 2010.
[2] R. Domer, Transaction level modeling of computation, Center for Embedded Computer Systems, University of California, Irvine, Tech. Rep. CECS-06-11, Aug. 2006.
[3] A. Gerstlauer, H. Yu, D. Gajski, "RTOS modeling for system-level design," *DATE*, Mar. 2003.
[4] H. Posadas, *et al.*, "RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model," *DAES*, 10(4), Dec. 2005.
[5] J.C. Prevotet, *et al.*, "A Framework for the Exploration of RTOS Dedicated to the Management of Hardware Reconfigurable Resources," *Reconfigurable Computing and FPGAs*, 2008.
[6] A. Bouchhima, *et al.*, "Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration," *ASPDAC*, Jan. 2005.
[7] G. Schirner, A. Gerstlauer, R. Dömer, "Fast and Accurate Processor Models for Efficient MPSoC Design," *TODAES*, 15(2), Feb. 2010
[8] J. Schnerr, O. Bringmann, A. Viehl, W. Rosenstiel, "High-performance timing simulation of embedded software," *DAC*, Jun. 2008.
[9] Z. Wang, A. Herkersdorf, "An efficient approach for system-level timing simulation of compiler-optimized embedded software," *DAC*, Jul.2009.
[10] Y. Hwang, S. Abdi, D. Gajski, "Cycle approximate retargettable performance estimation at the transaction level," *DATE*, Mar. 2008.
[11] A. Bouchhima, *et al.*, Automatic instrumentation of embedded software for high level hardware/software co-simulation, *ASP-DAC*, Jan. 2009.
[12] Z. Wang, *et al.*, An approach to improve accuracy of source-level tlms of embedded software, *DATE*, Mar. 2011.
[13] S. Stattelmann, *et al.*, Fast and accurate source-level simulation of software timing considering complex code optimizations, *DAC*, Jun. 2011.
[14] A. Pedram, D. Craven, and A. Gerstlauer, Modeling cache effects at the transaction level, *IESS*, Langenargen Germany, Sep. 2009.
[15] L. Gao, *et al.*, Multiprocessor performance estimation using hybrid simulation, *DAC*, Anaheim CA, Jun. 2008.
[16] M. Krause, *et al.*, Combination of instruction set simulation and abstract rtos model execution for fast and accurate target software evaluation, *CODES+ISSS*, Atlanta GA, Oct. 2008
[17] D. Sunwoo, *et al.*, Presto: An fpga-accelerated power estimation methodology for complex systems, *FPL*, Aug-Sep 2010.
[18] E. Copty, *et al.*, Transaction level statistical analysis for efficient microarchitecture power and performance studies, *DAC*, Jun. 2011.
[19] D. Brooks, V. Tiwari, and M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, *ISCA*, 2000.
[20] S. Li, *et al.*, McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures, *MICRO*, Dec. 2009.
[21] V. Tiwari, S. Malik, and A. Wolfe, Power analysis of embedded software: a first step towards software power minimization, IEEE Trans. VLSI Syst., vol. 2, pp. 437445, Dec. 1994.
[22] The architectural description language project, ver 2.0.0. [Online]. Available: http://opensource.freescale.com/fsl-oss-projects
[23] P. Razaghi, A. Gerstlauer, "Host-Compiled Multicore RTOS Simulator for Embedded Real-Time Software Developement," *DATE*, Mar. 2011.
[24] Open Virtual Platform [online]. Available: http://www.ovpworld.org
[25] P. Razaghi, A. Gerstlauer, "Automatic Timing Granularity Adjustment for Host-Compiled Software Simulation," *ASPDAC*, Jan. 2012.
[26] MiBench V.1.0 [online]. Available: http://www.eecs.umich.edu/mibench
[27] R. Dömer, *et al.*, "System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design," *EURASIP JES*, 2008(647953), 13 (2008).