Analyzing Power Consumption of Message Passing Primitives in a Single-chip Multiprocessor

Mirko Loghi
Università di Verona
37134 Verona, Italy
loghi@sci.univr.it

Luca Benini
Università di Bologna
40134 Bologna, Italy
lbenini@deis.unibo.it

Massimo Poncino
Università di Verona
37134 Verona, Italy
massimo.poncino@univr.it

Abstract

In this work we propose a methodology for the accurate analysis of the power consumption of interprocessor communication in a MPSoC, and the construction of high-level power macromodels.

The models leverage a complete MPSoC power estimation environment, that allows to evaluate the power consumption of various software functions, including message passing primitives, which could not be fully characterized in single-processor analysis framework developed in the past. Based on this data we built power macromodels that achieve average estimation errors below 5%.

1. Introduction

Multiprocessor systems-on-chip (MPSoCs) are becoming widespread in high-end consumer applications. One of the distinctive challenges in the evolution of MPSoCs, and a key differentiating factor with respect to traditional high-end, large scale multiprocessors, is the energy efficiency requirement. For instance, cellular handsets feature dual-processor SoCs (DSP and RISC micro-controller), and future mobile multimedia and ambient intelligence platforms will integrate a much larger number of processor cores [1]. Mobile multimedia and ambient intelligence systems are characterized by tight power budgets [2], which drive energy optimization efforts across the entire design flow, from technology to software.

Software power estimators are strategic for energy-aware software development. While several power estimators for single processor platforms have been developed in the recent past [6, 7, 9, 5], none of the published approaches is flexible and general enough to analyze the power consumed in a complex MPSoC with several communicating cores.

This work provides a first attempt in this direction, with three contributions. First, we developed a complete power estimation environment for MPSoCs, using technology-homogeneous and silicon-validated power models for all major hardware modules (processors, memories, communication links). Second, we performed a detailed and accurate analysis of the power consumption of interprocessor communication in a MPSoC setting. Third, we propose simple and accurate high-level power macromodels for the communication primitives of the underlying operating systems, which allow to correlate of power consumption with of high-level metrics of increasing levels of abstraction, ranging from the number of bus accesses to message size.

The accuracy of the models is quite good, especially for the very high-level ones, for which the average error is below 5%.

2. Related Work

Software power modeling has been actively studied in recent years. However, most of the research has been focused onto single-processor systems. The seminal work by Tiwari et al. [3] introduced the popular *instruction-level power analysis* approach, which builds a model associating power consumption to instructions or instruction pairs, based on a set of characterization experiments. Better accuracy can be achieved by *micro-architectural power models* [4, 5], which rely on the knowledge of the main functional units of a microprocessor.

All the above-mentioned approaches focus on the CPU; however, software execution consumes power also in the memory system, system buses, and peripherals. Several researchers [6, 7, 8, 9] proposed thus *full-system estimators*, that couple instruction set simulators with CPU, memory, bus and peripherals power models.

To avoid full instruction-level simulation, several techniques have been proposed for characterizing software power consumption at a coarser level of granularity. Macromodeling techniques have been proposed for sub-routine calls [10], for operating system calls [11, 12, 13], and even for entire tasks [14, 15]. The main advantage of these techniques is that they can provide early feedback on the power consumed by complex programs without the computational cost of instruction-level simulation.

A completely different class of approaches is based on an abstract representation of the system, in terms, for instance, of a queue network or a Petri net, where processors are modeled as requesters, and buses and memories as queues or places [16, 17, 18]. These approaches provide analytical performance models which are in principle applicable at a very high-level of abstraction, provided that the suitable parameters can be extracted from the inspection of the application.

3. Multiprocessor Platform

The simulation platform [19] used in this work consists of (i) a configurable number of 32-bit ARM processors, (ii) their private memories, (iii) a shared memory, (iv) a hardware interrupt module, (v) a hardware semaphore module, (vi) a 32-bit interconnect (ST Microelectronics' STBus). The interrupt device allows processors to send interrupt signals to each other, while the semaphore device implements *test-and-set* operations. Both these devices are mapped in the addressing space and are used for interprocessor communication and synchronization purpose.

The system is configurable in several of its components, such as the memory latencies, the number and the size of the caches, the topology of the bus, and many others. The specific instance of the platform used in this work consists of a four-CPU system, with (8KB I-cache, 4KB D-cache) memories, each with a two-cycle latency.

The platform provides cycle-accurate power models for its components [20], which are referred to the same technology (0.13 μm by STM). Since the simulation is cycle-accurate and the power models are invoked at each cycle, MPARM provides, on a cycle-by-cycle basis, the energy spent by all the components (core, cache, memories and bus).

The platform also includes a complete operating system and its support APIs, RTEMS [21], a light-weight OS suitable for embedded systems, which offers complete support for multiprocessing, and provides an API for inter-processor communication and synchronization. Applications can directly use the communication primitives provided by the API to implement parallel programs. The target of this work is exactly the characterization of these communication primitives.

4. RTEMS Interprocessor Communication

Characterizing the communication primitives requires a precise identification of what communication primitives are available in RTEMS. The communication architecture in RTEMS is structured into two layers:

- The top layer consists of a application-level API which is independent of the implementation of the low-level communication infrastructure (*Multiprocessor Communications Interface Layer (MPCI)*). This API is based on *message queues*.
- The bottom layer is the MPCI layer, which relies on a set of platform-specific procedures which enable the processors to communicate with each other. The MPCI manages a pool of buffers called *packets* and their transmission between system nodes. Packet buffers contain the messages sent between the nodes.

In RTEMS, the basic inter-thread communication primitives are *message queues*. Threads communicate by writing/reading *messages* into/from a queue using two system calls: `rtems_message_queue_send` and `rtems_message_queue_receive` (`send` and `receive`, for short).

4.1. Remote `send`

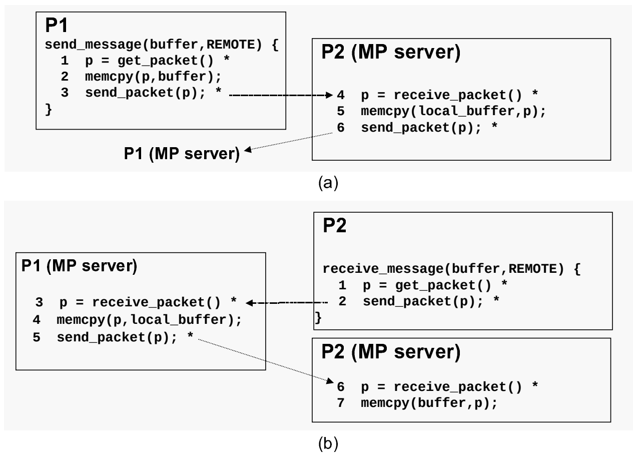
We assume that a thread T_1 (on processor P_1) `send`s a message onto a remote queue created on processor P_2, which is `receive`d by thread T_2 (on processor P_2). This is the list of operation executed by the `send`:

1. Using the global identifier of the queue, T_1 obtains a packet buffer using the `get_packet` primitive. The latter returns a pointer in the shared space which will be used by T_1 to access the queue. Synchronized accesses to this buffer are realized by means of *locks*, which can be thought of as an equivalent of a hardware *Test-and-Set*, implemented by polling a given location of the shared memory. The `get_packet` blocks in case the lock is used.
2. T_1 uses this pointer by filling it with the desired data. This is done by means of a call to the `memcpy()` function of the standard C library.
3. Once the shared buffer has been manipulated, T_1 sends the buffer to the queue, by means of the `send_packet` primitive. This operation is actually implemented as follows. First, it copies the actual data in an area of the shared memory which works as a communication channel between the calling threads and the queue. To access the buffer, it must first acquire a lock. After this has been done, it sends an interrupt; the relative ISR wakes up the *multiprocessing server* (i.e., the RTEMS module in charge of the management of the MPCI layer, MP server for short) by unlocking a semaphore on which the server was waiting. From now on, the processing happens on P_2.
4. On P_2, the MP server executes a `receive_packet`, which, after busy waiting on the lock, gets the address of the shared buffer described above.
5. The data are explicitly copied from this shared buffer to a local buffer (via `memcpy`).
6. Finally, the MP server sends a response (via `send_packet`) to the MP server on P_1, used as an acknowledge. Again, this operation requires busy waiting on a lock.

In this case, T_2 simply copies from the local buffer in the kernel space to a local buffer into its address space.

4.2. Remote `receive`

The remote `receive` (i.e a `receive` system call invoked on a remote queue) works in a similar manner, even if not fully symmetrically. The MP server on the processor which owmns the queue is in charge to fill the shared buffers for the requesting thread, which will copy (via `memcpy`) the data into its local memory.
Figure 1 summarizes the sequence of operations of the two primitives. Symbol "`*`" denotes operations which requires acquisition of a lock, while arrows between operations indicate actions triggered by interrupts.



```
send_message(buffer,REMOTE) {
  1  p = get_packet() *
  2  memcpy(p,buffer);
  3  send_packet(p); *
}
```

```
4  p = receive_packet() *
5  memcpy(local_buffer,p);
6  send_packet(p); *
```

```
receive_message(buffer,REMOTE) {
  1  p = get_packet() *
  2  send_packet(p); *
}
```

```
3  p = receive_packet() *
4  memcpy(p,local_buffer);
5  send_packet(p); *
```

```
6  p = receive_packet() *
7  memcpy(buffer,p);
```

Figure 1. Remote `send` (a) and `receive` (b).

5. Measuring Performance and Power

In order to evaluate the cost of the communication primitives we inserted an ARM-specific instruction (`SWI`, SoftWare Interrupt) into the OS source code. When the simulator identifies one of these instructions, it triggers the measuring related tasks. We have implemented several specific actions in RTEMS that used the special `SWI` mechanism to be activated. Four of them are strictly related to measurement functions:

- *start*: activates all counters that are used to accumulate all the energy and performance statistics.
- *stop*: deactivate the counters, thus stopping the gathering of data by disabling all counters. It must have a matching *start*.
- *clear*: resets all counter (but keeps them active). This feature is useful for activating measures of specific operations, so as to obtain *local* statistics.
- *dump*: writes out the data collected so far on a log file.

As an example, this is how the measurements is triggered for a generic `send_packet`.

```
start_metric();
  status = send_packet(p,REMOTE);
 stop_metric();
 dump_metric();
 clear_metric();
```

6. Quantitative Analysis

After the software probing has been inserted into RTEMS, we wrote the benchmarks for the characterization. In particular, we designed a synthetic benchmark relative to a four-processor configuration of the simulation platform and parameterized with respect to the message size (8, 16, 32, 64, 128 and 256 Bytes). In the benchmark, one of the processors sends a message to another one, while the other two generate tunable dummy traffic on the bus.
In the various runs, energy values are collected so that distributions of the energy consumption can be built. Figure 2 shows the plots of the distributions for the `send` and the `receive`.

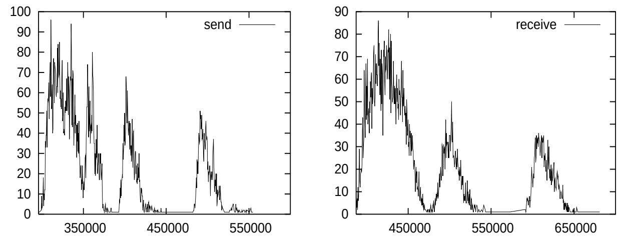


Figure 2. Distribution of Energy for `send` and `receive`.

We can notice the multi-modal nature of the distribution, with peaks corresponding to the various message sizes used in the characterization.
From the raw data, we derived three models, with parameters of different abstraction levels, corresponding the the following templates:

1. $E = E_0$; this model assumes a constant value for each `send` and `receive`. Although very abstract (it is sufficient to keep the count of the messages exchanged to obtain an estimation of the cost of communication), it has the worst accuracy.
2. $E = A \cdot N_{ba} + B \cdot T_{bus} + C$; this model depends on two low-level parameters, the number of bus accesses (N_{ba}), and the time spent for the whole communication task (T_{bus}). This model yields a very good accuracy, but it requires the knowledge of low-level quantities, which require a full cycle-accurate simulation to be extracted.
3. $E = \alpha \cdot S + \beta$; this model is based on one parameter only, the size of the exchanged messages (S). It has slightly worse accuracy than the previous model, but message size S can be easily extracted from the source code, without even requiring simulation.

Table 1 shows the parameters (in pJ) for the three models, related to `send` as well as to `receive`, obtained by running least-mean-square regression on the raw data (Models 2 and 3).
In order to measure the actual error of the models, we ran another benchmark, similar to the one used for characteriza-

tion, but with different message sizes (24, 30, 40, 100, 150 and 200 Bytes), and a different behavior of the traffic generator tasks.

	Model 1	*Model 2*			*Model 3*	
	E_0	A	B	C	α	β
`send`	373674	-456	199	18284	743	311250
`receive`	469589	-582	215	-42797	832	399718

Table 1. Coefficients of the Energy Models.

Table 2 shows the errors of the three models, measured against the energy values measured by cycle-accurate simulations.

	Average Error [%]		*Worst-Case Error [%]*	
	`send`	`receive`	`send`	`receive`
Model 1	10.11	8.36	25.42	27.59
Model 2	1.08	1.08	4.48	3.77
Model 3	4.79	4.63	10.45	17.89

Table 2. Estimation Relative Error.

Due to its high error, the *Model 1* does not provide good estimations and its use is limited to preliminary, rough evaluations. Conversely, *Model 2* supplies very accurate results, but it needs low-level parameters which require accurate simulations. *Model 3* relies on a high level parameter and fits very well the data. Even if some seldom worst-cases show errors above 10%, the average error is under 5% and this exhibits the good behavior of the model in the most common cases. This model can be successful used into the inner loop of a software optimization, because it can be used without resorting to simulation.

The surprisingly good result of *Model 3* is actually based on the statistical properties of the quantity under measure. Sending (and receiving) a message, in fact, requires a very large number of operation whose cost is nearly constant, while a minor fraction of operations depends on variable issues, such as the traffic on the bus. Furthermore the number of operations, whose cost depends on the traffic, is strictly related to the message size, so the unpredictability is further reduced.

7. Conclusions

We have proposed a first solution towards high-level modeling of software energy consumption in MPSoCs. We focused on the consumption of the interprocessor communication primitives provided by the operating system.

Leveraging a cycle-accurate simulation framework which accurately models the hardware as well as the software architecture of the system, we devised a novel energy measuring strategy based on software probes, which allows to precisely monitor the energy of selected operations.

The energy values thus measured have been used to extract an empirical energy macro-model which provides average errors below 5%.

References

[1] E. Aarts, R. Roovers, ”IC Design Challenges for Ambient Intelligence,” *Design, Automation and Test in Europe*, pp. 3-7, 2003.

[2] L. Benini, M. Poncino, ”Ambient Intelligence: A Computational Platform Perspective” in: *Ambient Intelligence: Impact on Embedded System Design*, T. Basten, M. Geilen, H. de Groot eds. Kluwer Academic Publishers, 2003.

[3] V. Tiwari, S. Malik, A. Wolfe, ”Power Analysis of Embedded Software: a First Step Towards Software Power Minimization,” *IEEE Transactions on VLSI Systems*, Vol. 2, no. 4, pp. 437-445, Dec. 1994.

[4] D. Brooks et al., ”Power-Aware Micro-Architecture: Design and Modeling Challenges for Next-Generation Microprocessors,” *IEEE Micro*, Vol. 20, no. 6, pp. 24-44, Nov.-Dec. 2000.

[5] N. Vijaykrishnan et al.”Evaluating Integrated Hardware-Software Optimizations using a Unified Energy Estimation Framework,” *IEEE Transactions on Computers*, Vol. 52, no. 1, pp. 59-76, Jan. 2003.

[6] T. Simunic, L. Benini, G. De Micheli, “Energy-Efficient Design of Battery-Powered Embedded Systems,” *IEEE Transactions on Very Large-Scale Integration Systems*, Vol. 9, no. 1, pp. 15–28, Feb 2001.

[7] S. Gurumurthi, et al. ”Using Complete Machine Simulation for Software Power Estimation: the SoftWatt Approach,” *International Conference on High-Performance Computer Architecture*, pp. 124-133, 2002.

[8] J. Henkel, Y. Li, ”Avalanche: an Environment for Design Space Exploration and Optimization of Low-Power Embedded Systems,” *IEEE Transactions on VLSI Systems*, Vol. 10, no. 4, pp. 454-468, Aug. 2002

[9] T. Givargis, F. Vahid. J. Henkel, ”Instruction-Based System-Level Power Evaluation of System-on-a-Chip Peripheral Cores,” *IEEE Transactions on VLSI Systems*, Vol. 10, no. 6, pp. 856-863, Dec. 2002.

[10] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, ”Cosimulation-Based Power Estimation for System-on-Chip Design,” *IEEE Transactions on VLSI Systems*, Vol. 10, no. 3, pp. 253-266, June 2002.

[11] T. Tan, A. Raghunathan, N. Jha, ”Embedded Operating System Energy Analysis and Macro-Modeling,” *International Conference on Computer Design*, pp. 515-222, 2002.

[12] A. Acquaviva, L. Benini, A. Ricco’, ”Energy Characterization of Embedded Real-Time Operating Systems,” in L. Benini, M. Kandemir, J. Ramanujam, *Compilers and Operating Systems for Low Power*, Kluwer Academic Publishers 2003.

[13] R. Dick, G. Lakshminarayana, A. Raghunathan, N. Jha, ”Analysis of Power Dissipation in Embedded Systems using Real-Time Operating Systems,” *IEEE Transactions on CAD*, Vol. 22, no. 5, pp. 615-627, May 2003.

[14] R. Dick, N. Jha, ”MOGAC: a Multi-Objective Genetic Algorithm for Hardware-Software co-synthesis of distributed embedded systems,” *IEEE Transactions on CAD*, Vol. 17, no. 10, pp. 920-935, Oct. 1998.

[15] A. Acquaviva, E. Lattanzi, A. Bogliolo, L. Benini, ”A Simulation Model for Streaming Applications over a Power Manageable Wireless Link,” *European Simulation and Modeling Conference*, Oct. 2003.

[16] “System-level Performance Analysis for Designing On-Chip Communication Architectures,” K. Lahiri, A. Raghunathan, S. Dey, *IEEE Transactions on CAD*, Vol. 20, No. 6, June 2001, pp. 768–783.

[17] “SPI - A System Model for Heterogeneously Specified Embedded Systems,” D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, J. Teich, *IEEE Transactions on VLSI Systems*, Vol. 10, No. 4, August 2002, pp. 379–389.

[18] “A Formal Approach to MpSoC Performance Verification,” K. Richter, M. Jersak, R. Ernst, *IEEE Computer*, Vol. 36, No. 4, April 2003, pp. 60–67.

[19] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, R. Zafalon, “Analyzing On-chip communication in a MPSoC Environment”, *DATE’04: Design, Automation and Test in Europe*, Feb. 2004, pp. 752–757.

[20] M. Loghi, M .Poncino, L. Benini, “Cycle-Accurate Power Analysis for Multiprocessor Systems-on-a-Chip”, *GLSVLSI’04: Great Lake Symposium on VLSI*, Apr. 2004 pp. 401–406.

[21] RTEMS home page, `www.rtems.com`.