Software Architectural Transformations: A New Approach to Low Energy Embedded Software

T. K. Tan[†], A. Raghunathan[‡], and N. K. Jha[†]

[†] Dept. of Electrical Eng., Princeton University, NJ 08544

[‡] NEC Labs, Princeton, NJ 08540

Abstract— *Previous work on software optimization for low energy has focussed on instruction-level optimizations and compiler techniques. We argue, and demonstrate, that significant energy savings could be "left on the table" if energy is not considered during the design of the* software architecture*. As a first step towards addressing this gap, we propose a systematic framework for software architectural transformations to reduce energy consumption.*

We consider software architectural transformations in the context of the multi-process software style driven by an operating system (OS), which is very commonly employed in energy-sensitive embedded systems. Our methodology for applying software architectural transformations consists of: (i) constructing a software architecture graph representation, (ii) deriving initial energy and performance statistics using a detailed energy simulation framework, (iii) constructing sequences of atomic software architectural transformations, guided by energy change estimates derived from high-level energy macro-models, that result in maximal energy reduction, and (iv) generation of program source code to reflect the optimized software architecture. We employ a wide suite of software architectural transformations whose effects span the application-OS boundary, including how the program functionality is structured into architectural components (e.g.*, application processes, signal handlers, and device drivers), and connectors between them (inter-component synchronization and communication mechanisms).*

We present experimental results on several multi-process embedded software programs, in the context of an embedded system that features the Intel StrongARM processor and the embedded Linux OS. The presented results clearly underscore the potential of the proposed methodology (up to 66.1% reduction in energy is obtained). In a broader sense, our work demonstrates the impact of considering energy during the earlier stages of the software design process.

I. Introduction

Low power design techniques have been investigated in the hardware design domain at various levels of the design hierarchy. Fig. 1 presents the different levels of hardware design

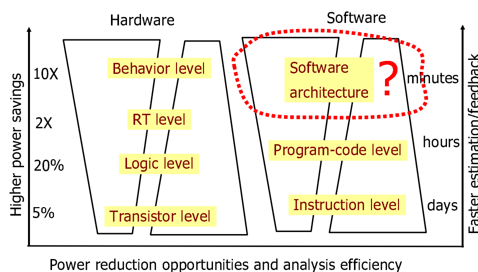

Fig. 1. Analysis and optimization efficiency at different levels of design abstraction

abstraction, and illustrates that the efficiency of analysis, and the amount of power savings obtainable, are much larger at higher levels [1–3]. It is natural to hypothesize whether such an observation can be extended to the software design domain. In the software design domain, low power techniques have been extensively investigated at the instruction level, and through enhancements to various stages of the high-level programming language compilation process. However, if we consider the assembly instruction level for software to be analogous to the logic level for hardware, and software compilation to be analogous to logic synthesis, it is clear that there exists a "gap" at the software architecture level, where very little, if any, research has been conducted on energy reduction.

There are two major aspects to consider at the software architecture level of the design hierarchy:

- The first aspect concerns the selection of a design style, generally called the *software architectural style*. Several software architectural styles are known, which provide designers with interesting tradeoffs between efficiency, design effort, portability, maintainability, *etc.*
- The second aspect concerns the design of a *concrete software architecture* within the framework of the selected architectural style.

We believe that both of the above considerations may have significant impact on the overall energy consumption of the embedded system. In this paper, we focus on the impact of concrete software architecture design on energy consumption. We consider various *software architectural transformations* that affect how the program functionality is structured into architectural components, as well as the connectors among them [4]. The specific software architectural style that we use for our illustrations consists of multi-process applications being executed in the run-time environment provided by an embedded OS. We call this an OS-driven multi-process software architectural style. The architectural components correspond to software entities such as application processes, signal handlers, device drivers, *etc.*, while connectors include inter-process communication (IPC) and synchronization mechanisms.

In the next section, we highlight our contributions in this paper and discuss some related work. Section III, we describe our software architectural transformation methodology in detail. Section IV describes the experimental method used to evaluate our techniques, and presents the experimental results. Section V presents the conclusions.

II. Contributions and Related Work

In this section, we highlight our contributions and discuss some related work.

A. Paper Contributions

The major contributions of this paper are as follows:

- We propose a systematic methodology for applying software architectural transformations, which consists of: (i) constructing a software architecture graph to represent a software program, (ii) deriving an initial profile of the energy consumption and execution statistics using a detailed energy simulation framework, (iii) evaluating the energy impact of atomic software architecture transformations through the use of energy macro-models, (iv)

Acknowledgments: This work was supported by DARPA under contract no. DAAB07-02-C-P302.

constructing sequences of atomic transformations that result in maximal energy reduction, and (v) generation of program source code to reflect the optimized software architecture.
- We show how to use OS energy macro-models to provide energy change estimates that predict the effect of various software architectural transformations.
- We experimentally demonstrate the utility of our techniques in the context of several typical programs running on an embedded system that features the Intel StrongARM processor and embedded Linux as the OS. Significant energy savings (up to 66.1%) were achieved through the energy-efficient software architectures generated by the proposed techniques.

B. Related Work

System-level software energy reduction strategies usually involve some kind of resource scheduling policy, whether it is scheduling of the processor or the peripherals. In complex embedded systems, this often centers around the adaptation of the OS. Some general ideas on adapting software to manage energy consumption were presented in [5–7]. Vahdat *et al.* [8] proposed revisiting several aspects of the OS for potential improvement in energy efficiency. Lu *et al.* [9] proposed and implemented OS-directed power management in Linux and showed that it can save more than 50% power compared to traditional hardware-centric shut-down techniques. Bellosa [10] presented thread-specific online analysis of energy-usage patterns that are fed back to the scheduler to control the CPU clock speed.

Software architecture has been an active research area for the past few years in the real-time software and software engineering communities, with emphasis on software understanding and reverse engineering. Many common software architectural styles are surveyed in [4]. Wang *et al.* [11] evaluated the performance and availability of two different software architectural styles. Carrière *et al.* [12] studied the effect of connector transformation at the software architectural level in a client-server application. None of these studies considered the effect of the software architecture on energy consumption. Our work, on the other hand, provides a systematic framework for harnessing software architectural transformations to minimize energy consumption.

IMEC's work on embedded software synthesis [13] considers synthesis of real-time multi-threaded software based on a multi-thread graph (MTG) model. They focus on static scheduling of the threads without the use of an operating system. They do not target energy consumption. Our work also considers multi-process embedded software, but emphasizes the energy-efficient use of the operating system through proper software transformations.

III. Energy Minimization through Software Architectural Transformations

In this section, we describe a methodology for minimizing the energy consumption of embedded software through software architectural transformations. Section III-A provides an overview of the entire flow, while Sections III-B through III-D detail the important aspects.

A. Overview of the Software Architectural Energy Minimization Methodology

The proposed software architecture level energy minimization flow is shown in Fig. 2. In the figure, the cylinders represent the entities to be operated upon, and the rectangles represent the steps in the algorithm. The methodology

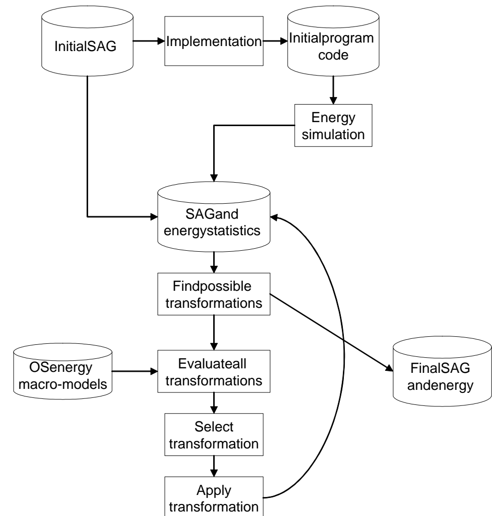

Fig. 2. The software architecture level energy minimization methodology

starts with an initial software architecture, represented as a *software architecture graph* or SAG (as described in Section III-B). The energy consumption E_0 of the initial software architecture is obtained by compiling the corresponding program source code into an optimized binary for the target embedded system, and by profiling this implementation using a detailed energy simulation framework.[1] The execution and energy statistics collected from this step are subsequently used to guide the application of software architectural transformations. Our methodology optimizes the software architecture by applying a sequence of atomic software architectural transformations, or moves, that lead to maximum energy savings. These transformations are formulated as transformations of the SAG, and are described in detail in Section III-D. We use an iterative improvement strategy to explore sequences of transformations. Selection of a transformation at each iteration is done in a greedy fashion. That is, at each iteration, we choose from all the possible transformations the one that yields maximum energy reduction. The iterative process ends when no more transformation is possible.

During the iterative process, for each architectural transformation considered, we need to estimate the resulting change in energy consumption. Since this estimation is iterated a large number of times, it needs to be much more efficient than hardware or instruction-level simulation. We utilize high-level energy macro-models to provide energy change estimates, as described in Section III-C.

After an energy-efficient software architecture is obtained (as an SAG), the program source code is generated to reflect the optimized software architecture. The optimized program code can be executed in the low-level energy simulation framework to obtain accurate energy estimates for the optimized software architecture.

The remainder of this section details the important steps in the proposed methodology.

B. Software Architecture Graph

We represent the architecture of an embedded software program as an SAG. An example of an SAG, for a program

[1] Note that the energy simulation framework needs to simulate the OS together with the application, since the effects of the software architectural transformations span the application-OS boundary.

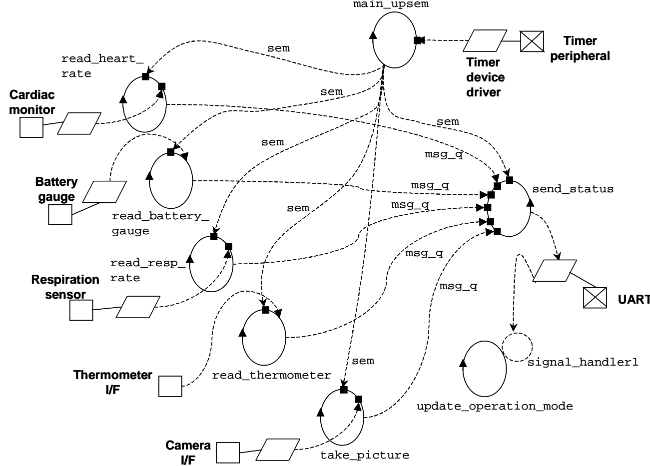

Fig. 3. Software architecture graph for a situational awareness subsystem

employed in a situational awareness system, is depicted in Fig. 3. In the SAG, vertices represent hardware or software entities. Vertices can be of several types:

- Hardware devices are represented by an empty box, with an optional crossbar for active devices (*e.g.*, the UART peripheral in Fig. 3).
- Device drivers are represented by a parallelogram (*e.g.*, the timer device driver in Fig. 3).
- Application processes are represented by an ellipse with an arrow on the perimeter (*e.g.*, process `read_heart_rate` in Fig. 3).
- Signal handlers are represented by a dotted circle attached to the corresponding application process (*e.g.*, `signal_handler1`, which is attached to process `update_operation_mode` in Fig. 3).

The association between hardware devices and device drivers is depicted by a solid line connecting them. Arrowed edges between any two vertices represent the communication of data or control messages, and are annotated with the selected IPC mechanisms. Since the system is OS-driven, they are to be implemented as calls to the system functions. For example, the edge from process `main_upsem` to process `read_heart_rate` in Fig. 3 is a control message that is implemented using the semaphore service provided by the OS. Naturally, an edge from or to a hardware device represents the transfer of data using OS system functions. A small solid square at the termination of an edge indicates a blocking communication. Otherwise, the communication is assumed to be non-blocking.

C. Energy Modeling at the Software Architecture Level

We denote a specific software architecture configuration for a given embedded software program as C. The energy consumption of this software architecture for a fixed amount of computation (application functionality) is denoted by $E(C)$. The energy consumption of the initial architecture C_0 is denoted by $E_0 = E(C_0)$. In an absolute sense, for a given embedded system, $E(C)$ depends on various parameters related to how the software architecture is translated into the final executable implementation (*e.g.*, compiler optimizations used). The energy consumption of each software architecture could be accurately evaluated by specifying it as program source code, feeding it through the software compilation flow, and using a detailed system simulation framework. However, this would be too time-consuming to use in the context of an automatic software architectural transformation framework, since each candidate architecture could require hours to evaluate. In our context, the following observations are worth noting:

- We are only interested in comparing the inherent energy efficiency of different software architectures without regard to the subsequent software implementation process.
- We perform a (one-time) detailed energy profiling of the initial software architecture. The initial architecture is modified by applying a sequence of atomic transformations. Hence, we only require *energy change estimates*, *i.e.*, estimates of the difference in energy consumption before and after the application of each atomic software architectural transformation.
- The transformations utilized in our methodology do not affect the "core functionality" of an application. Rather, they affect energy consumption by altering the manner in which OS services are employed. As shown later, this implies that we can use high-level OS energy macro-models to evaluate the energy impact of software architectural transformations.

Given a software architecture configuration C_1, and a transformation $T_{C_1 \rightarrow C_2}$, a new configuration C_2 is created. The equation relating the energy consumption of these two configurations is:

$$E(C_2) - E(C_1) = \Delta E(T_{C_1 \rightarrow C_2}) \quad (1)$$

where $\Delta E(T_{C_1 \rightarrow C_2})$ denotes the energy change incurred by performing transformation $T_{C_1 \rightarrow C_2}$.

As mentioned earlier, the effect of software architectural transformations is to alter the manner in which OS services are employed during execution of the application. Hence, we are able to estimate ΔE using high-level energy macro-models specific to the OS. An energy macro-model is a function (*e.g.*, an equation) expressing the relationship between the energy consumption of a software function and some predefined parameters. In the above-mentioned context, these energy macro-models are called the OS energy characteristic data [14]. OS energy macro-models proposed in [14] show on average 5.9% error with respect to the energy data used to obtain the macro-models. This level of accuracy is sufficient for relative comparison of different transformations.

Basically, the OS energy characteristic data consists of the following:

- *The explicit set*: These include the energy macro-models for those system functions that are explicitly invoked by application software, *e.g.*, IPC mechanisms. Given these energy macro-models and execution statistics collected during the initial system simulation, the energy change due to an atomic software architectural transformation can be computed. For example, suppose the energy macro-models for two IPC's, IPC_1 and IPC_2, are given by:

$$E_{ipc1}(x) = c_{11} + c_{12}x \quad (2)$$
$$E_{ipc2}(x) = c_{21} + c_{22}x \quad (3)$$

where x is the number of bytes being transferred in each call, and $c's$ are the coefficients of the macro-models. The amount of energy change incurred by replacing IPC_1 with IPC_2 is given by

$$\Delta E = [E_{ipc2}(x) - E_{ipc1}(x)]\, N_{ipc} \quad (4)$$
$$= [(c_{21} - c_{11}) + (c_{22} - c_{12})x]\, N_{ipc} \quad (5)$$

where N_{ipc} is the number of times this IPC is invoked in the specific application process under consideration. The values of parameters such as N_{ipc} and x are collected during the detailed profiling of the initial software architecture. The energy changes due to other system function replacements can be calculated similarly.

- *The implicit set*: These include the macro-models for the context-switch energy, timer-interrupt energy and rescheduling energy. In particular, the context-switch energy, E_{ctx}, is required to calculate the energy change due to process merging. E_{ctx} is defined to be the amount of round-trip energy overhead incurred every time a process is switched out (and switched in again) [14].

In the next section, we introduce a set of atomic software architectural transformations that we have selected. Computation of the energy changes incurred by these transformations is facilitated by the availability of the OS energy macro-models.

D. Software Architectural Transformations Employed in the Proposed Methodology

Since we have adopted the OS-driven multi-process software architectural style, the moves that we consider are mainly manipulations of the components (application processes, signal handlers, and device drivers), and the connectors (inter-process synchronization and communication mechanisms). Some of the specific transformations presented here, including manipulation of application processes or process structuring, have been investigated in other related areas such as software engineering, mostly in a qualitative manner. For example, a good introduction to process structuring can be found in [15]. In this sub-section, we formulate a wide range of atomic software architectural transformations and show how they can be systematically applied as transformations of the SAG. For each atomic transformation, we also include a brief analysis of the energy change incurred. Note that some of these atomic transformations may not directly result in large energy reduction, but may enable other moves that reduce more energy eventually. Also, note that the inverse of any move discussed here is also a valid move.

D.1 Temporal cohesion driven process merging

This transformation involves merging of two software processes that are driven by events whose instances have a one-to-one correspondence. A special case is periodic events that occur at the same rate. Application of this transformation decreases the number of processes by one. This transformation is illustrated in Fig. 4. The energy change due to this transformation is estimated as:

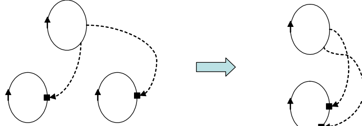

Fig. 4. Temporal cohesion driven process merging

$$\Delta E = -E_{ctx} N_{ctx} \quad (6)$$

where E_{ctx} is the context switch energy explained above, and N_{ctx} is the number of times each of the processes is activated. Note that this transformation can be collated with other transformations such as code computation migration and IPC merging to further reduce energy, as illustrated later.

D.2 Sequential cohesion driven process merging

This transformation involves merging of two software processes that are executed sequentially because one process passes data or control to the other. It decreases the number of processes by one and also removes the IPC between them. This transformation is illustrated in Fig. 5. The energy change due to this transformation is estimated as:

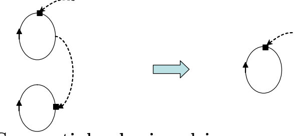

Fig. 5. Sequential cohesion driven process merging

$$\Delta E = -E_{ipc}(x) N_{ipc} - E_{ctx} N_{ctx} \quad (7)$$

where $E_{ipc}(x)$ is the total IPC (read and write) energy for communicating x amount of data, N_{ipc} is the number of times the IPC between the two processes is invoked, and N_{ctx} is the number of times each of the processes is activated.

D.3 Intra-process computation migration

This transformation involves moving some of the code in a process so that two IPC writes (or sends) can be replaced by a single write (or send). It exploits the capability of IPC system calls that can accept multiple messages (with potentially different destinations), *e.g.*, the `msg_snd()` function in Linux. It is useful in reducing the constant overhead involved in invoking the IPC function. It is illustrated in Fig. 6. The energy change due to this transformation is estimated as:

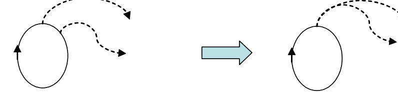

Fig. 6. Computation migration

$$\Delta E = -\left(E_{ipc_wr}(x) + E_{ipc_wr}(y) - E_{ipc_wr}(x+y)\right) N_{ipc_wr} \quad (8)$$

where E_{ipc_wr} refers to IPC write energy, N_{ipc_wr} is the number of times one of the IPC writes is invoked, and x and y represent the average amount of data transferred per call through the first and second IPC writes, respectively. Note that this transformation enables further IPC merging in some cases.

D.4 Temporal cohesion driven IPC merging

This transformation involves merging of two IPC's that are driven by the same event. This move is illustrated in Fig. 7. The energy change due to this transformation is estimated as:

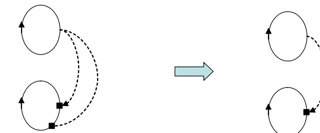

Fig. 7. Temporal cohesion driven IPC merging

$$\Delta E = -E_{ipc_read}(x) N_{ipc_read} \quad (9)$$

where E_{ipc_read} is the IPC read energy for x amount of data and N_{ipc_read} is the number of times one of the IPC reads is invoked.

D.5 IPC replacement

This transformation involves replacing the IPC mechanism associated with an edge in the SAG by another functionally equivalent mechanism. For example, message passing can be replaced by shared memory with semaphore protection. This transformation is illustrated in Fig. 8. The energy change

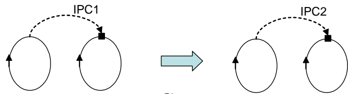

Fig. 8. IPC replacement

due to this transformation is estimated as:

$$\Delta E = [E_{ipc2}(x) - E_{ipc1}(x)]\, N_{ipc} \quad (10)$$

where E_{ipc1} and E_{ipc2} refer to the energy of the first and second IPC mechanism, respectively, and N_{ipc} is the number of times this IPC is invoked.

D.6 System function replacement

System functions initially used in the application program may not be the most energy-efficient choices under the specific context in which they are used. In this case, replacement of these system functions by lower energy alternatives leads to energy reduction. This transformation is illustrated in Fig. 9. The energy change due to this transformation is

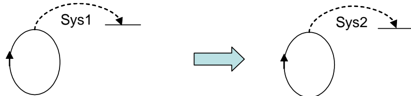

Fig. 9. System function replacement

estimated as:

$$\Delta E = [E_{sys2} - E_{sys1}]\, N_{sys} \quad (11)$$

where E_{sys1} (E_{sys2}) is the energy of system function $sys1$ ($sys2$) and N_{sys} is the number of times the system function is invoked.

D.7 Process embedding as signal handlers

This advanced transformation embeds a process as a signal handler into another process. By doing so, the number of processes is reduced by one, reducing context switch related energy. However, there is an overhead due to signal handling. This transformation is illustrated in Fig. 10. The dashed

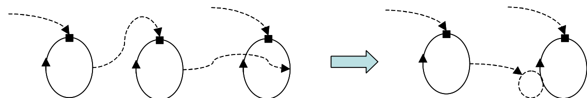
Fig. 10. Process embedding

circle attached to the second process of the transformed SAG represents the signal handler of the second process. The energy change due to this transformation is estimated as:

$$\Delta E = -E_{ctx}N_{ctx} + E_{sig}N_{sig} \quad (12)$$

where N_{sig} is the number of times the signal handler, being the replacement of the embedded process, is activated. E_{sig} is the signal handling energy.

D.8 Computation migration to device drivers

This advanced transformation migrates some of the preliminary computation from the process interfacing with an active device into the device driver for the device. By doing so, data transfer initiated by the active device can be potentially reduced in its quantity or frequency. A common example is device buffering, wherein data are aggregated within the device driver and sent to the application process in larger chunks, reducing the rate at which the device driver activates the application process. This transformation is illustrated in Fig. 11. As one can see, this transformation requires changes

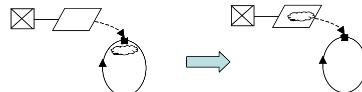
Fig. 11. Computation migration to device drivers

to the device driver. In conventional general-purpose systems, a clear separation is advocated between application functionality and low-level drivers or system software. Nevertheless, for embedded systems, such changes are usually permissible. The energy change due to this transformation is roughly estimated as:

$$\Delta E = E_{ctx} \sum_i \left(N^{(2)}_{ctx,i} - N^{(1)}_{ctx,i} \right) \quad (13)$$

where $N^{(1)}_{ctx,i}$ ($N^{(2)}_{ctx,i}$) is the number of times process i is activated before (after) the change. Note that since the reduction in process activations may further propagate beyond the process directly interfacing with the device driver, we take the sum of reductions in process activations across all the processes in the SAG.

IV. Experimental Results

As a proof of concept, we applied our software architecture energy minimization methodology to various multi-process example programs designed to run under the embedded Linux OS, specifically `arm-linux v2.2.2` [16]. The benchmarks include: `Aware` – the situational awareness system shown in Fig. 3, `Headphone` – a program used in an audio headset application, `Vcam` – embedded software from a video camera recorder, `Climate` – a telematics application for collecting and processing climate information, `Navigator` – software from a global navigation system, and `ATR` – a part of an automatic target recognition program. The hardware platform for our experiments was based on the Intel StrongARM embedded processor [17]. We used EMSIM [18] as the detailed energy simulation framework in our experiments. With detailed system models, EMSIM allows execution of the Linux OS within the simulated environment. The high-level energy macro-models of the Linux OS, which were used to guide the application of software architectural transformations, were obtained from [14].

Table I shows the experimental results. Major columns 2 and 3 show the details of the original and optimized software programs, respectively. # proc denotes the number of processes. # ipc denotes the number of IPC's involved. Major column 4 shows the energy reduction as a percentage of the original energy consumption. Significant energy reductions (up to 66.1%, average of 25.1%) can be observed for the examples. Note that the energy savings obtained through software architectural transformations are largely independent of, and complementary to, energy savings obtained through lower-level optimizations, including compiler techniques.

In the next sub-section, we illustrate the application of architectural transformations to a simple software program.

TABLE I

Experimental results showing the energy impact of software architectural transformations.

Examples	Original			Optimized			% Energy reduction
	Energy (mJ)	# proc	# ipc	Energy (mJ)	# proc	# ipc	
`Aware`	12.956	8	11	8.204	7	9	36.7%
`Headphone`	1.668	6	8	1.461	3	2	12.4%
`Vcam`	1.572	4	5	1.375	2	1	12.5%
`Climate`	0.239	4	5	0.081	2	1	66.1%
`Navigator`	1.659	5	7	1.456	3	3	12.2%
`ATR`	6.940	4	7	6.199	3	4	10.7%

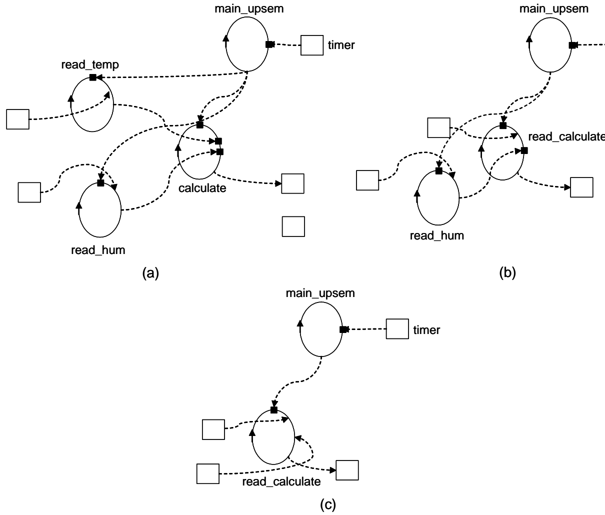

Fig. 12. Sequence of software architectural transformations for an example system

A. Software Architecture Level Energy Minimization: A Case Study

We consider the `climate` system from Table I. An SAG that represents the initial software architecture for this program is shown in Fig. 12(a). Explanation of the annotations in the figure were provided in Section III-B. For the purpose of illustration, we consider this program to execute under the embedded Linux OS. Edges coming from the hardware devices (indicated as empty square boxes) indicate the data flow from the devices to the application processes, and are implemented using `read()` system calls in Linux. Some of these reads are blocking, indicated by a small square box at the tip of the arrow.

The initial software architecture program code is written in C. The energy consumption statistics of this program are obtained by running it in the energy simulation framework EMSIM [18]. Under the initial software architecture, the energy consumption of the entire system for three iterations of program execution is $0.239mJ$. That is, $E_0 = 0.239mJ$. The first two transformations merge the software processes `read_temp` and `calculate` as well as the IPC edges that involve them, resulting in the `read_calculate` process. The new SAG is shown in Fig. 12(b). Note that the evaluation of the transformations is performed with the help of OS energy macro-models, as described in Section III-C. The next two transformations merge `read_hum` and `read_calculate` as well as the IPC edges that involve them. The final SAG is shown in Fig. 12(c).

The transformed program is re-constructed from the final software architecture, and simulated again in the EMSIM energy simulation framework. The energy consumption estimate is obtained to be $E = 0.081mJ$, which corresponds to an energy savings of 66.1%.

V. Conclusions

Energy minimization of embedded software at the software architecture level is a new field that awaits exploration. As a first step in this direction, we presented a systematic methodology to optimize the energy consumption of embedded software by performing a series of selected software architectural transformations. As a proof of concept, we applied the methodology to a few multi-process example programs. Experiments with the proposed methodology has demonstrated promising results, with energy reductions up to 66.1%. We believe that software architecture level techniques for energy reduction can significantly extend and complement existing low power software design techniques.

References

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

[2] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*, Kluwer Academic Publishers, Norwell, MA, 1997.

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

[4] D. Garlan and M. Shaw, "An introduction to software architecture," Tech. Rep. CMU-CS-94-166, School of Computer Science, Carnegie-Mellon Univ., Jan. 1994.

[5] J. R. Lorch and A. J. Smith, "Software strategies for portable computer energy management," *IEEE Personal Communications*, vol. 5, no. 3, pp. 60–73, June 1998.

[6] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *Proc. ACM Symp. Operating System Principles*, Dec. 1999, pp. 48–63.

[7] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson, "Quantifying the energy consumption of a pocket computer and a Java Virtual Machine," in *Proc. SIGMETRICS*, June 2000, pp. 252–263.

[8] A. Vahdat, A. Lebeck, and C. S. Ellis, "Every Joule is precious: The case for revisiting operating system design for energy efficiency," in *Proc. 9th ACM SIGOPS European Workshop*, Sept. 2000.

[9] Y. H. Lu, L. Benini, and G. De Micheli, "Operating-system directed power reduction," in *Proc. Int. Symp. Low Power Electronics & Design*, July 2000, pp. 37–42.

[10] F. Bellosa, "The benefits of event-driven energy accounting in power-sensitive systems," in *Proc. ACM SIGOPS European Workshop*, Sept. 2000.

[11] W. L. Wang, M. H. Tang, and M. H. Chen, "Software architecture analysis - A case study," in *Proc. Annual Int. Computer Software & Applications Conf.*, Aug. 1999, pp. 265–270.

[12] S. J. Carrière, S. Woods, and R. Kazman, "Software architectural transformation," in *Proc. 6th Working Conf. Reverse Engineering*, Oct. 1999.

[13] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli, "Software synthesis for real-time information processing systems," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds., Chap. 15, pp. 260–296. Kluwer Academic Publishers, Boston, MA, 1994.

[14] T. K. Tan, A. Raghunathan, and N. K. Jha, "Embedded operating system energy analysis and macro-modeling," in *Proc. Int. Conf. Computer Design*, Sept. 2002, pp. 515–522.

[15] H. Gomaa, *Software Design Methods for Real-time Systems*, Addison-Wesley, Boston, MA, 1993.

[16] ARM Linux, http://www.arm.linux.org.uk/.

[17] Intel Corporation, *Intel StrongARM SA-1100 Microprocessor Developer's Manual*, Aug. 1999.

[18] T. K. Tan, A. Raghunathan, and N. K. Jha, "EMSIM: An energy simulation framework for an embedded operating system," in *Proc. Int. Symp. Circuit & Systems*, May 2002, pp. 464–467.