Energy-Optimizing Source Code Transformations for OS-driven Embedded Software

Yunsi Fei†, Srivaths Ravi‡, Anand Raghunathan‡, and Niraj K. Jha†
† Dept. of Electrical Engg., Princeton University, NJ 08544, Email: {yfei,jha}@ee.princeton.edu
‡ NEC Labs, Princeton, NJ 08540, Email: {sravi,anand}@nec-lab.com

Abstract

The increasing software content of battery-powered embedded systems has fueled much interest in techniques for developing energy-efficient embedded software. Source code transformations have previously been considered for application software to reduce its energy consumption. For complex embedded software applications, which consist of multiple concurrent processes running with the support of an embedded operating system (OS), it is known that the OS and the application-OS interaction significantly affect energy consumption. However, source code transformations explicitly targeting these effects have not been sufficiently studied.

This paper proposes novel transformations for the source code of OS-driven multi-process embedded software programs in order to reduce their energy consumption. The key features of our optimizations are that they span process boundaries, and that they minimize the energy consumed in the execution of OS functions and services — opportunities which are beyond the reach of conventional compiler optimizations and source code transformation techniques. We propose four types of transformations, namely process-level concurrency management, message vectorization, computation migration and inter-process communication mechanism selection. We discuss how to systematically identify opportunities for the proposed transformations and apply them directly to the program source code.

We have applied the proposed techniques to several multi-process software benchmark programs, and evaluated their applicability in the context of an embedded system containing an Intel StrongARM processor and embedded Linux OS. Our techniques achieve up to 37.9% (23.8% on an average) energy reduction compared to highly compiler-optimized implementations.

I. Introduction

Limited battery life has made energy efficiency a critical issue for mobile computers and portable embedded systems, such as laptops, PDAs, cell phones, *etc.* Recent studies have examined the architecture of the overall system for energy saving opportunities, which consider not only hardware components for energy reduction, but also energy-efficient software design and compilation [1]. Low energy software design can be performed at three levels of abstraction: instruction level, program or source-code level, and algorithm level [2]. Instruction-level techniques [3] have focused on efficient code generation for a program using energy consumption as the design metric, register allocation to minimize memory access overheads, and instruction reordering to reduce inter-instruction overheads, *etc.* While these approaches can be automated in the compilation process, the overall energy consumption savings is small, and is strongly tied to the processor architecture. Algorithmic approaches, on the other hand, achieve significant energy savings through careful selection of the algorithms used in the software [4], [5]. Since these approaches are mostly based on human intuition and knowledge, significant manual effort is essential for them. In contrast, program code restructuring approaches achieve the best balance between energy efficiency and automation. Their impact on energy consumption tends to be high since they work with a global view of the program. In addition, such techniques are platform-independent, making them easily portable to different architectures. In this work, we propose novel source code transformations that can reduce program energy consumption.

Several source code transformations proposed for reducing software energy consumption have been surveyed in [6], [7]. These techniques do not consider the effects of the OS, and are applicable to source code within a single process[1]. More importantly, conventional source code transformation techniques do not target the coarse-grained system-level concurrency and global data flow among multiple processes. Our work is based on the fact that, in concurrent multi-process programs, factors such as inter-process synchronization, data communication, context switches, OS intervention, *etc.*, can significantly affect the overall energy consumption of an application. Therefore, we propose several OS-driven source code transformations that reduce the energy overheads associated with the aforementioned factors as follows.

- Concurrent processes can be managed in an energy-efficient manner through merging or splitting of processes (*process-level concurrency management*).
- Inter-process communication (IPC) versus process memory usage trade-offs can be exploited for energy reductions through buffering of communicated data (*message vectorization*).
- Computations in one process can be migrated to another process to reduce the number of IPC and data volume (*computation migration*).
- The IPC mechanism (shared memory, pipe, message queue, *etc.*) used for a given communication channel in an application can be selected in cognizance of the communication characteristics of that channel (*IPC mechanism selection*).

We experimentally demonstrate the efficacy of the above transformations in the context of several multi-process programs running on a single-processor embedded platform. The platform features the Intel StrongARM processor and embedded Linux as the OS. The proposed transformations achieve energy savings up to 37.9% (23.8% on an average) compared to traditional compiler optimizations.

II. Related work

There have been several studies that have analyzed the impact of the OS on the energy consumption of software and adapted the OS for low energy. Dick *et al.* [8] first developed an energy profiler for applications executing on an embedded system based on the Fujitsu SPARCLite processor running μC/OS II. Tan *et al.* [9] developed an energy simulator, called EMSIM, for an embedded system featuring a StrongARM processor that uses embedded Linux as its OS. Vahdat *et al.* [10] re-examined the design and implementation of OSs by taking energy consumption as the primary metric instead of performance. Lu *et al.* [11] implemented an OS-directed power management scheme in Linux and achieved significant power reductions compared to hardware-centric shutdown techniques. Pillai *et al.* [12] proposed a class of real-time dynamic voltage scaling algorithms and modified the OS's real-time scheduler and task management services to provide significant energy savings. These works mostly focus on redesigning or modifying the OS to achieve energy savings. Few studies have examined the usage of source-level software transformations to reduce the energy overheads associated with OS intervention.

Software synthesis for embedded systems with OSs is another area wherein the issues of task-level concurrency and data parallelism have been examined. Given a set of concurrent processes specified in a language called FlowC, Cortadella *et al.* [13] provided a procedure for extracting tasks from the processes, and scheduling their execution efficiently on a single processor. This approach is useful in low-end embedded systems, where there is limited OS support and hence, the software itself has to explicitly provide many of the OS services. Sgroi *et al.* [14] and Thoen *et al.* [15] provided similar static scheduling algorithms. Prayati *et al.* [16] explored the issues of extracting concur-

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

[1] The term *process* denotes the OS notion of a basic concurrent unit of execution, with its own associated address space and other resources needed for its execution.

rency between tasks, performing scheduling in the presence of various constraints and mapping the tasks to different processors in a multiprocessor system. They also proposed guidelines for improving the concurrency of the applications considered. Software architectural transformations were proposed as a means of reducing energy consumption in [17], based on optimizing an abstract software representation from which the program implementation is subsequently generated.

Other approaches [18], [19] focus on optimizing the communication overheads associated with concurrent processes that are specified in a concurrent language called ERLANG. They do not study the effect of such transformations on energy consumption. Several source code transformations have also been proposed to address the data transfer and storage costs [20] and cache effects [2] in applications.

III. Preliminaries: Embedded system software model

In this section, we describe the embedded system software model used in this work. We begin our discussion with an overview of multi-process embedded systems and then present an abstracted system-level view of the embedded software called *control/data flow process network*, which is the software model used as the starting point for the proposed optimizations.

A. The control/data flow process network

We consider a multi-process embedded system to be specified as a set of concurrently communicating sequential processes that is implemented on a specific single-processor platform with some real-time requirements. For each process, a set of input and output ports are defined, and point-to-point communication occurs between processes through channels between ports [13], [17]. The system communicates with the environment (for example, the disk, network adapter card, *etc.*) through some input and output ports with no channel defined. We refer to these ports as the *primary input* and *primary output* ports of a process, and the hardware devices (in the environment) connected to them are called *sources* and *sinks*, respectively. We classify the primary input ports into two classes: *controllable* (connected to a passive source) and *uncontrollable* (connected to an active source). Controllable ports, as the name indicates, are under the control of the software. Thus, objects can be read from them at any given time when the system performs an "acquire" operation (*e.g.*, the access of a file on a disk by the system when it issues a "read" command). Uncontrollable ports are under the control of the active environment, which sends objects to the system through the ports. This implies that the system must always be ready to receive objects from them and react accordingly by performing operations (*e.g.*, a web-server which responds to the client requests). At least one process is required for each uncontrollable input port to react to the event on that port. We only consider the uncontrollable ports as primary inputs.

We consider a hierarchical software representation called *the control/data flow process network*. This model captures a process-level view of the software at the top level, in which only essential control/data flow and dynamic constructs (*e.g.*, semaphores, IPC, *etc.*) are visible. The process network also associates with each process a more fine-grained view as specified by its function call graph. The software is profiled to provide this view as well as the various statistics necessary for the proposed inter-process optimizations.

Fig. 1 shows the control/data flow process network for an example embedded software program that consists of five processes $P_0 - P_4$. A process is represented by an oval, and the call graphs corresponding to each process is also available (as seen for processes P_1 and P_2). Processes communicate with each other through unidirectional data communication channels, which are represented by solid directed arcs (*e.g.*, $channel_1$, $channel_2$, $channel_3$ in Fig. 1). The small black diamonds in each process represent the ports for communication. Each arc is annotated with a $(volume/\#IPC)$ tuple (profiling-generated statistics) that indicates the data volume communicated for a single communication instance ($volume$) as well as the number of such communications ($\#IPC$). These data are available for both inter- and intra-process communications.

External hardware devices ($source_1$, $source_2$, $sink$) in the process network are shown as grey boxes. Control flow among processes is shown as dashed arcs, and the synchronization mechanism (*semaphore*) is labeled along the arcs.

B. Control/data flow process network extraction

In this work, we focus on software programs written in the C programming language, which is a non-concurrent imperative language.

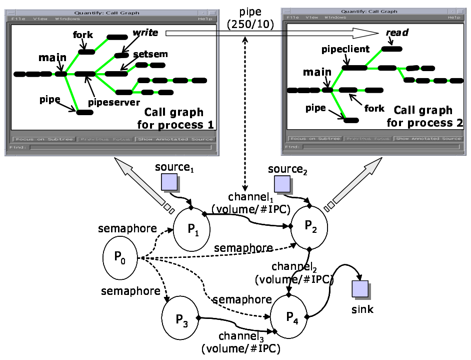


Fig. 1. A control/data flow process network

We consider the POSIX interface [21], since it is supported by a wide range of popular OSs including Linux and various versions of Unix. However, the proposed approach and techniques are fairly general, and are also applicable to other programming languages and OSs.

Starting from the software source code, we derive the process network by (a) defining the boundary of each process, (b) locating the IPC implementation as well as the communication between the process and its outer environment, and (c) determining the control flow between processes (synchronization). The process network model for a multi-process embedded system software can be automatically generated by modifying advanced compilers such as IMPACT [22] and SUIF2 [23]. Since there are no generic communication primitives in the C language and POSIX, we restrict our attention to specific implementations of IPC, namely, *pipe*, *message passing* and *shared memory*, which are three commonly used mechanisms for data communication. For the pipe mechanism, a set of two file descriptors is declared globally and a pipe connecting these two file descriptors is created. The two file descriptors in this illustration correspond to the communication ports of a process in the control/data flow process network. The pipe corresponds to the channel between two processes with data flow directed from the *send* process to the *receive* process. In this way, the *read/write* pair in the source code defines the IPC. Instances of system calls, *msgsnd()* and *msgrcv()*, that operate on the same message queue define the communication pair for the message passing mechanism. The *send* process sends a message with a specific type and fixed size to the queue. The *receive* process then selects the appropriate message by matching its type. In the case of shared memory, the IPC is implemented through an update of the shared data region by one process followed by the access of the data by other processes. Therefore, explicit synchronization is typical of shared memory IPC so as to avoid race conditions between communicating processes.

In addition to the IPC mechanisms described above, any read and write operations that are not part of a communication pair (as in IPC) correspond to communication between a process and its environment (*i.e.*, process communication from/to a *source*/*sink*, as shown in Fig. 1). In this way, we can identify all the ports, channels, synchronization between processes, primary inputs/outputs, and the corresponding sources/sinks needed for a control/data flow process network.

IV. Transformation techniques for inter-process optimization

In this section, we illustrate the various transformation techniques and investigate their application to a few examples.

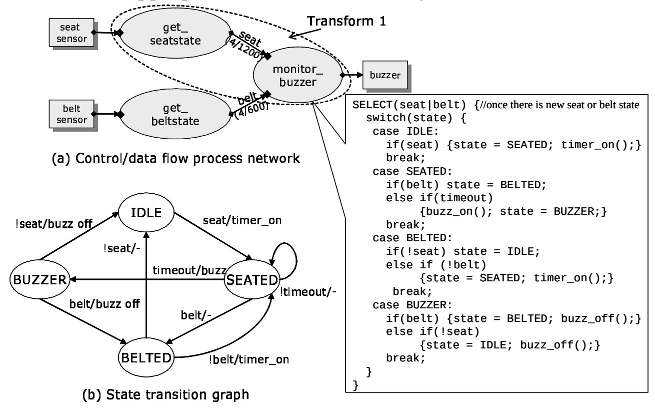
A. Process-level concurrency management

The basic objective of process-level concurrency management is to ensure that the number of concurrent processes is minimized to reduce the intervention of the underlying OS (factors such as context switch, synchronization and data communication between processes), while requiring each process to be efficient (*e.g.*, in terms of memory usage).

We will now motivate with an example how merging of processes can be effective in reducing energy costs.

Example 1: Consider an embedded system that provides seat belt alerts in cars. Fig. 2(a) shows the control/data flow process network for this example. Two sensors (*seat sensor* and *belt sensor*) collect seat and belt status data continuously and drive two processes, $get\_seatstate$ and $get\_beltstate$. Another process, $monitor\_buzzer$, acquires seat and belt status data from these two processes and monitors the state of the system. A state transition graph is described in Fig. 2(b), and the pseudo-code is given in the box on the right.

The system starts in the state marked IDLE. When the seat is taken, it enters into the SEATED state, and a timer is turned on. If the seat belt is not fastened until the timer expires, a buzzer is turned on to alert the driver, and the system enters the BUZZER state. Otherwise when the belt is fastened in time, the system switches to the BELTED state. When the driver releases the seat belt or leaves the car, the system transits to the appropriate states automatically.



```
SELECT(seat|belt) {//once there is new seat or belt state
  switch(state) {
   case IDLE:
     if(seat) {state = SEATED; timer_on();}
     break;
   case SEATED:
     if(belt) state = BELTED;
     else if(timeout)
          {buzz_on(); state = BUZZER;}
     break;
   case BELTED:
     if(!seat) state = IDLE;
     else if (!belt)
          {state = SEATED; timer_on();}
      break;
   case BUZZER:
     if(belt) {state = BELTED; buzz_off();}
     else if(!seat)
          {state = IDLE; buzz_off();}
     break;
  }
}
```

Fig. 2. An example of applying process merging transformation

If we examine the process network to identify opportunities for process merging, we can see that

- Processes $get\_seatstate$ and $monitor\_buzzer$ communicate 1,200 times and each IPC corresponds to a data traffic of four bytes. Thus, the two processes are candidates for merging (we refer to this transformation as $Transform$ 1).
- Processes $get\_beltstate$ and $monitor\_buzzer$ are also candidates for merging (we refer to this transformation as $Transform$ 2).
- Processes $get\_seatstate$ and $get\_beltstate$ are both connected to active devices implying that merging of the two processes would result in a process that can potentially miss certain input events (say, while blocking on one active device, input data from another active device may be lost). Therefore, these two processes cannot be merged.

In both $Transform$ 1 and $Transform$ 2, the number of concurrent processes is decreased by one, and one IPC channel is removed. Consequently, the IPCs on this channel are also removed and the context switch between processes decreases. Thus, we can expect the energy consumption to be reduced in both the configurations.

Table I validates the above hypothesis by showing energy consumption results for the original and merged configurations. The energy simulator EMSIM [9] was used to generate the results. The energy data also show that *Transform 1* is better than *Transform 2*. This is expected, since the overall inter-process data volume is significantly reduced in the former case compared to the latter (4,800 bytes as opposed to 2,400 bytes). ■

TABLE I

COMPARISON BETWEEN ORIGINAL AND TRANSFORMED SOURCE CODE FOR THE SEAT-BELT EXAMPLE

Source code	# proc	# channels	Total energy (mJ)	Reduction (%)
Original	3	2	24.27	-
Transform1	2	1	18.02	25.8%
Transform2	2	1	19.62	19.2%

The above example provides a few general guidelines for process merging, namely:

1. Two processes *P* and *Q* can be merged if the the number of active devices connected to the resulting merged process does not exceed one, *i.e.*, either *P* and *Q* are driven by the same active device, or only one of them is driven by an active device.
2. Two processes *P* and *Q* are good candidates for merging if the number of IPCs is high and the communicated data volume is high.

B. Message vectorization

This transformation vectorizes the messages communicated between two processes, thereby reducing the number of IPCs. We now illustrate this transformation using the following example.

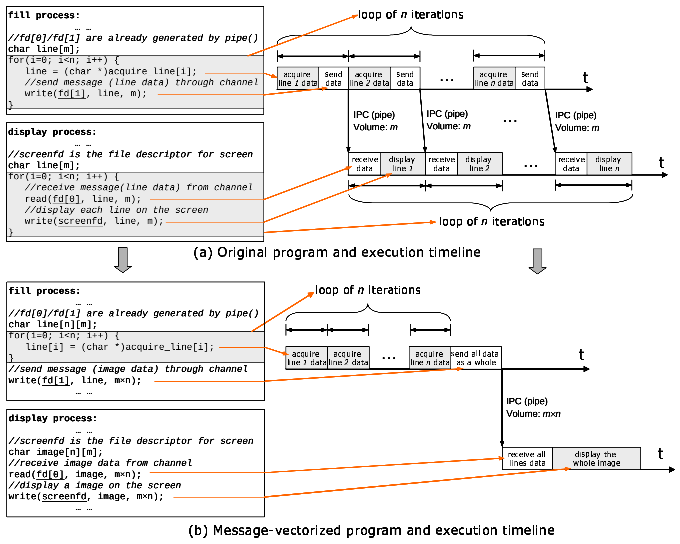
Example 2: Consider the *DrawLine* application that is used in many graphics support systems. *DrawLine* consists of two main processes, *fill* and *display*, which are shown in Fig. 3(a). The *fill* process generates the lines of an image, and passes each line of data to the *display* process that is connected to a graphical display device. The lines are then written line-by-line to the display memory until the image is completed. The IPC mechanism in this application is a pipe.

Fig. 3(a) also shows the execution profile for the two processes. The *fill* process contains a loop with *n* iterations. In each iteration, it acquires a line and sends the data right away. The *display* process also contains a loop with *n* iterations. In each iteration, the process blocks and waits for the line from the *fill* process and writes it to the display memory. Assuming the image has *n* lines, and the length of each line is *m*, there are *n* messages (each of size *m*) sent from the *fill* process to the *display* process.

An alternative software implementation is achieved by applying message vectorization to this application. Fig. 3(b) shows the transformed source code and the new execution profiles of the two processes. In the *fill* process, the messages are first vectorized or buffered. They are then transmitted to the *display* process in a burst. The transformed code has the following properties.

- The total volume of messages (which corresponds to the image size) passed between the two processes remains the same as before.
- The number of messages, as well as the number of sends and receives, are reduced significantly (to one in this case).
- The memory usage of the two processes increases since the array size has increased from m to $m \times n$.

Thus, vectorization will remain effective so long as the energy overheads of buffering do not outweigh the gains of reducing the number of messages communicated.



```
fill process:
            … …
//fd[0]/fd[1] are already generated by pipe()
char line[m];
for(i=0; i<n; i++) {
   line = (char *)acquire_line[i];
   //send message (line data) through channel
   write(fd[1], line, m);
}
```

```
display process:
            … …
//screenfd is the file descriptor for screen
char line[m];
for(i=0; i<n; i++) {
   //receive message(line data) from channel
   read(fd[0], line, m);
   //display each line on the screen
   write(screenfd, line, m);
}
```

(a) Original program and execution timeline

```
fill process:
            … …
//fd[0]/fd[1] are already generated by pipe()
char line[n][m];
for(i=0; i<n; i++) {
   line[i] = (char *)acquire_line[i];
}
//send message (image data) through channel
write(fd[1], line, m×n);
            … …
```

```
display process:
            … …
//screenfd is the file descriptor for screen
char image[n][m];
//receive image data from channel
read(fd[0], image, m×n);
//display a image on the screen
write(screenfd, image, m×n);
            … …
```

(b) Message-vectorized program and execution timeline

Fig. 3. Applying message vectorization to the *DrawLine* application

TABLE II

COMPARISON BETWEEN ORIGINAL AND OPTIMIZED SOURCE CODE FOR THE DRAWLINE APPLICATION

Source code	Total energy(mJ)	# proc	# channels	# IPCs
Original	8.24	2	1	1024
Optimized	7.17	2	1	1
Reduction	13.0%	0	0	1023

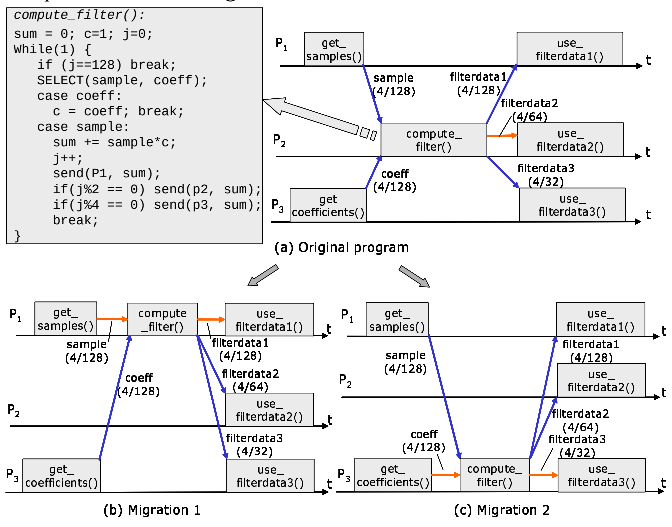
Table II summarizes the energy consumption results for the two cases. Due to message vectorization, the number of IPCs is reduced from 1,024 to 1. The total energy consumption is reduced by 13.0%, thus demonstrating the efficacy of this transformation. ■

C. Computation migration

This transformation relocates computations from one process to another so that the energy overheads due to synchronization and IPC get reduced. In the following example, we examine the basic notion of computation migration, and the various issues associated with it.

Example 3: Fig. 4(a) shows the execution profile of a signal processing application with three processes, P_1, P_2 and P_3. The parallel lines represent the execution time-lines of the three processes. Functions $get\_samples()$ and $get\_coefficients()$ (in processes P_1 and P_3, respectively) generate data $sample$ and $coeff$ that are inputs to function $compute\_filter()$ in process P_2. Function $compute\_filter()$ processes these data and generates data $filterdata1$, $filterdata2$ and $filterdata3$, which are used subsequently by functions $use\_filterdata1$, $use\_filterdata2$ and $use\_filterdata3$. Edges between the different functions indicate the IPC or intra-process communication described above. In addition, each edge is annotated with an (x/y) tuple, where x denotes the average per-communication data volume and y indicates the number of communications. For example, we can see from the profile that, on an average, four bytes of $sample$ are transmitted in a communication from $get\_samples()$ to $compute\_filter()$ and that this data communication occurs 128 times.

Consider the computation corresponding to $compute\_filter()$ in process P_2. The function sums the data obtained from P_1 ($sample$), which are weighted by the coefficients ($coeff$) from P_3, and filters the results at different rates to the three processes. In the figure, we can see that each computed value (sum) is sent to P_1 as $filterdata1$ so that the total number of IPCs from P_1 to P_2 is 128 (*i.e.*, data are filtered at the full rate). However, the results are filtered to P_2 at half that rate. Since $use\_filterdata2()$ and $compute\_filter()$ are in the same process, there is no IPC occurring, implying that the number of intra-process communications is 64. Finally, results are filtered to P_3 at a quarter rate resulting in 32 IPCs from P_2 to P_3.



```
compute_filter():
sum = 0; c=1; j=0;
While(1) {
   if (j==128) break;
   SELECT(sample, coeff);
   case coeff:
     c = coeff; break;
   case sample:
     sum += sample*c;
     j++;
     send(P1, sum);
     if(j%2 == 0) send(p2, sum);
     if(j%4 == 0) send(p3, sum);
     break;
}
```

Fig. 4. An example of computation relocation among processes

Assume that all the IPC mechanisms implemented in this example are the same. For any specific mechanism, consider the energy macro-model, which is given by the equation.

$$E = \alpha + \beta \times x \tag{1}$$

In Equation (1), x represents the number of bytes communicated, α represents the base energy consumption of a single IPC and β is the extra energy consumed per byte.

For a given process P_i, we can estimate the energy costs due to IPC with respect to a given computation C in any other process as follows. Let x_s be the total amount of source operand data that P_i contributes to C, x_r the amount of result data of C that is consumed by P_i, n_s the number of source data communications, and n_r the number of result data communications. Then, Equation (1) can be modified to yield the overall IPC cost for a process P_i and a computation C (not in P_i) as follows.

$$E_i = \alpha \times (n_s + n_r) + \beta \times (x_s + x_r) \tag{2}$$

For example, if we consider process P_1 and computation $compute\_filter()$, then the various parameters in Equation (2) are available from Fig. 4(a). Table III lists the relevant information for the three processes.

TABLE III

INTER-PROCESS AND INTRA-PROCESS COMMUNICATION STATISTICS FOR THE THREE PROCESSES P_1, P_2 AND P_3 WITH RESPECT TO FUNCTION $compute\_filter()$

Process	n_s	x_s $(bytes)$	n_r	x_r $(bytes)$	$n_s + n_r$	$x_s + x_r$ $(bytes)$
P_1	128	512	128	512	256	1,024
P_2	0	0	64	256	64	256
P_3	128	512	32	128	160	640

Consider now the question of migrating the computations in $compute\_filter()$ to either process P_1 (termed MIGRATION1) or P_3 (termed MIGRATION2), instead of remaining in P_2 (termed ORIGINAL). For the three configurations, we can evaluate the total energy costs using the energy macro-model given in Equation (2) for three different IPC mechanisms (pipe, message passing and shared memory). The actual α and β values are provided in Table V later. From the results shown in Table IV, we can see that irrespective of the IPC mechanism, MIGRATION1 is the most energy-efficient.

In order to understand the results, let E_j denote the IPC cost of process P_j for the $compute\_filter()$ computation. In this example, $E_1 > E_3 > E_2$ irrespective of the IPC mechanism used since P_1 has the largest $n_s + n_r$ and $x_s + x_r$ values and P_2 has the smallest $n_s + n_r$ and $x_s + x_r$ values. Assuming that relocating the computation part to different processes only induces different IPC energies, and the other energies remain the same, we can only compare the IPC energies as a first order of approximation. For the ORIGINAL case, the IPC cost is $E_{\text{ORIGINAL}} = E_1 + E_3$. When computation is relocated to P_1 (MIGRATION1) as shown in Fig. 4(b), the IPC cost becomes $E_{\text{MIGRATION1}} = E_2 + E_3$. Similarly, when relocated to P_3, the IPC cost is $E_{\text{MIGRATION2}} = E_1 + E_2$. Thus, $E_{\text{ORIGINAL}} > E_{\text{MIGRATION2}} > E_{\text{MIGRATION1}}$, implying that relocating computation to P_1, as shown in Fig. 4(b), should be the most energy-efficient transformation. Table IV verifies the results by comparing the energy consumption obtained using EMSIM between original and transformed source code. For each version of source code, the three different IPC mechanisms are implemented. ■

D. IPC mechanism selection

IPC selection refers to the process of choosing the most energy-efficient IPC mechanism (from among the alternatives supported by the OS) for each IPC. In this section, we discuss scenarios wherein one IPC mechanism is energy-wise more efficient than another, supported by energy macro-models for three popular IPC mechanisms in embedded Linux.

In order to derive the energy macro-models, we created three test applications. Each application consists of a communication pair with one shared memory, one message queue, and one pipe, respectively. Each mechanism has its own implementation for one-time IPC, as stated in Section III-B. The communicated data remain the same in each case. We parameterized each application in terms of the number of IPC operations executed by enclosing the IPC operation in a loop body. Since our objective is to build an energy macro-model for the different IPC mechanisms, we first obtained the energy estimates using the EMSIM simulator, for each IPC mechanism, and for varying number of IPC operations and fixed data volume of each IPC. We then studied the dependency of IPC energy costs on the inter-process data volume. This is because communication complexity (not just IPC frequency) affects performance and energy consumption.

Using the above approach, we derived the energy macro-model for the three IPC implementations shown in Table V.

The macro-model consists of two terms. The first term corresponds to a base or constant cost, which reflects the energy overhead due to a single IPC, while the second term denotes the energy variation with data volume. Generally, when no synchronization or memory protection method is employed, shared memory tends to be very energy-efficient (as shown in the first row of Table V). The main reason is that shared memory update and access (IPC) do not involve additional OS system calls, while the other two mechanisms require OS support for implementing send and receive. Note that, without protection, shared memory IPC can induce inconsistency when multiple processes have write access to the shared memory. Therefore, we implement a semaphore based synchronization method for shared memory, and the second row of Table V shows the corresponding IPC energy macro-model. After considering the energy consumed by synchronization,

TABLE IV

COMPARISON BETWEEN THE ORIGINAL AND TRANSFORMED SOURCE CODE FOR THE COMPUTATION MIGRATION EXAMPLE

Source code configuration	Total energy (mJ)			# proc	# IPC	volume of IPC data ($bytes$)
	Pipe	Shared Mem	Message			
ORIGINAL	36.26	37.97	39.33	3	416	1,664
MIGRATION1	**28.08**	**29.43**	**30.74**	3	224	896
MIGRATION2	30.99	31.17	32.33	3	320	1,280
Reduction	22.6%	22.5%	21.8%	0	192	768

we observed that shared memory IPC is similar to the other two mechanisms in terms of energy consumption.

TABLE V

ENERGY MACRO-MODELS FOR DIFFERENT IPC MECHANISMS IN LINUX

IPC mechanisms	**Macro-models** (nJ) (with x bytes)
shared memory	$E = 67.3 + 71.33x$
shared memory (with semaphore protection)	$E = 2294.5 + 71.33x$
pipe write/read	$E = 1892.6 + 2.45x$
msgsnd/msgrcv	$E = 2518.6 + 4.41x$

We also observed that pipe tends to be more energy-efficient compared to message passing, possibly due to the overheads associated with message copying and message queue handling (*e.g.*, keeping the messages in the queue, type matching, *etc.*).

V. Case Studies

This section presents the results of applying the proposed transformations to two example software applications: an underwater navigation control system [24] and an Ethernet packet processing system [8].

A. Methodology

We employ a simple design flow for systematically applying the various transformations in the case studies. Starting from the original C source code, we first profile and extract the control/data flow process network, as described in Section III. For the suite of transformations proposed in this paper, we determine the best candidate transformation by using the IPC energy macro-models described in Section IV-D as well as the energy macro-models for other explicit OS services in the source code (obtained from [25]). We then select the candidate transformation that has the least energy costs, apply it to the source code, and repeat the above steps for the modified source code. When no further energy savings are possible, the flow terminates.

In both the case studies, our flow considers all possible process merging opportunities. For message vectorization and computation migration, we evaluate a designer-specified set of buffer sizes and candidate computation/destination process pairs, respectively. For IPC mechanism selection, we consider the scenario when pipe or message passing or shared memory is exclusively used in the application.

For the results presented in the following sections, we use the energy simulator EMSIM [9] to estimate the energy consumption of an application (original and transformed source codes) running on an Intel StrongARM processor under an embedded Linux OS. The simulator executes on a 550 MHz Pentium Pro III Linux workstation with 256 MB memory.

B. Case study: An underwater navigation system

We optimize the underwater vehicle navigation system designed in [24]. Fig. 5(a) shows the control/data flow process network derived from its specification. Four external devices, *GPS receiver, Pressure sensors, Speed sensors* and *Altitude sensors* collect various raw data periodically, which are consumed by the four processes *Handle GPS protocol*, *Estimate depth*, *Estimate speed* and *Estimate Altitude*. These processes then estimate the approximate position parameters given by fix, depth, speed, and altitude. The *Main process* performs the role of a central monitor and triggers the four processes. Two other processes, namely, *Coordinate transform* and *Integrate velocity*, operate on the estimated data, and generate the approximate system position for display on the navigation screen. We have annotated the edges of the process network shown in Fig. 5(a) with the IPC statistics generated through profiling. The original source code uses shared memory as its IPC mechanism.

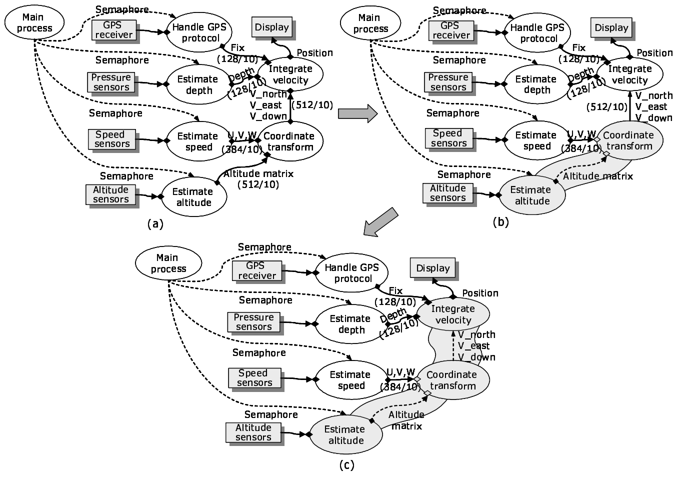


Fig. 5. The control/data flow process network for the underwater vehicle navigation system and example process merging transformations

We applied the transformations framework described in Section V-A to the original source code of the system, and determined a sequence of transformations that can best reduce the energy consumption, shown as the line on the left in Fig. 6. The transformations performed are as follows. Two process merging transformations (PM) are performed first, as shown in Figs. 5(b) and 5(c). The first transformation merges processes *Estimate altitude* and *Coordinate transform*, while the second merges the resultant process with *Integrate velocity* into a single process. The two process merging transformations reduce the overall IPC data volume by 61.5% and achieve significant energy savings (nearly 29.6%). Also, the number of processes is reduced to a minimum. The process merging transformations are followed by three message vectorization (MV) instances interleaved with one IPC mechanism selection (REPL). The IPC mechanism selection (REPL) results in pipes replacing the shared memory mechanism. Note that no computation migration is required for this case study. The energy reduction corresponding to the final optimized source code is 37.9% with the number of processes reduced to 5 (from 7), number of IPCs reduced to 3 (from 50) and IPC data volume reduced by 61.5%.

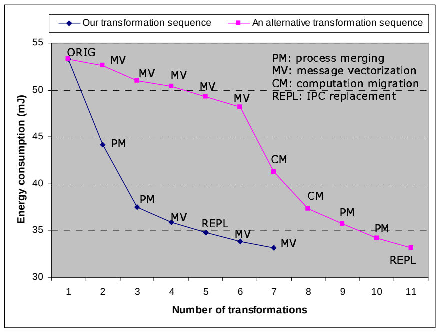


Fig. 6. Comparison of two different sequences of transformations

Fig. 6 also compares the optimization profiles of the above sequence of transformations with an alternative ordered sequence of transformations. For the alternative sequence, we used the following order: message vectorization followed by computation migration, process merging and IPC mechanism selection. The figure shows that the final transformed code in both the cases achieve similar energy savings. However, the number of transformations needed in our framework is smaller than the alternative scenario. In general, our experiments suggested that considering process merging transformations first resulted in the best energy savings with the least number of transformations, since process merging tends to have the largest impact on energy, and also indirectly influences the IPC (IPCs between merged processes are eliminated).

C. Case study: Ethernet packet processing system

The Ethernet packet processing system [8] is used at the lowest level of a TCP/IP based protocol stack. The system listens to the network ports, receives incoming packets, derives the checksum for the packet data and processes the packet header for transmission information. The packets are subsequently transmitted to the output devices periodically, driven by a timer.

Fig. 7(a) shows the most direct implementation of this system with three processes. Process *Receive packet* waits for incoming packets. Upon receiving a packet, it hands the packet over to the process labeled *Checksum & header handling*. Subsequently, the packet (with the checksum field filled) is passed to the *Transfer packet* process, which dispatches the packet to output devices. This implementation is very responsive, but each packet induces IPC twice, as annotated in the figure, which is expensive in terms of energy and execution time.

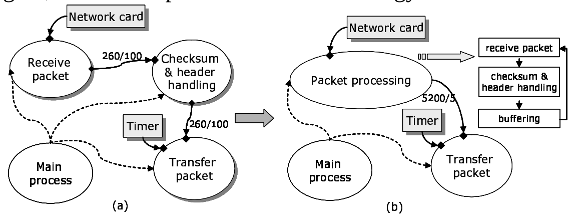


Fig. 7. The top-level control/data flow process network for the original and transformed implementations of an Ethernet packet processing system

Fig. 7(b) shows the transformed control/data flow process network for the system derived by our framework. The transformations used are process merging and vectorization, and the corresponding process-level statistics and energy consumption data for the original and transformed source codes are listed in Fig. 8(a). Process merging results in a single process, *Packet processing*, that encapsulates the computations corresponding to processes *Receive packet* and *Checksum & header handling*. In addition, the process buffers a packet into a fixed-size buffer after the checksum is computed. When the buffer is full, all the buffered packets are transferred to process *Transfer packet*. Thus, the number of IPCs among the processes is reduced greatly. IPCs are implemented with pipes in both the original and transformed cases.

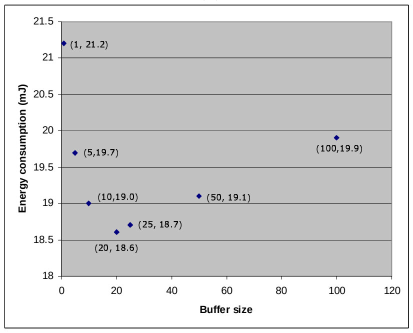
We also analyzed the effect of buffer size on overall energy consumption. Fig. 8(b) plots the energy consumption variation for various buffer sizes. We observed that with a buffer size of 20 packets, we can achieve the highest energy reduction (27.9%) compared to the original implementation. We also observe the trade-off between system memory usage and IPC overhead from this figure.

VI. Conclusions

In this work, we explored the OS-driven interface between processes in embedded software applications and proposed a novel set of source code transformation techniques that reduce energy consumption. We manage process-level concurrency through process merging to save context switch overhead and IPC. We modify the process interface by vectorizing the communications between processes and selecting an energy-efficient IPC mechanism. Finally, we also attempt to relocate computations from one process to another so as to reduce the number and data volume of IPCs. This set of transformations provides complementary optimization strategies to traditional compiler optimizations for energy savings.

Source code	#proc	#channel	#IPC	IPC volume (bytes)	Energy *(mJ)*
Original	4	2	200	52000	25.8
Process merging	3	1	100	26000	21.2
Message vectorization	3	1	5	26000	18.6

(a)



(b)

Fig. 8. (a) Energy consumption for original and transformed source codes, and (b) energy consumption variation with buffer size (after process merging)

References

[1] L. Benini and G. De Micheli, "System-level power optimization techniques and tools," *ACM Trans. Design Automation Electronics Systems*, vol. 5, no. 2, pp. 115–192, Apr. 2000.

[2] E.-Y. Chung, L. Benini, and G. De Micheli, "Source code transformation based on software cost analysis," in *Proc. Int. Symp. System Synthesis*, Sept.-Oct. 2001, pp. 153–158.

[3] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Processing Systems*, vol. 13, no. 2(3), pp. 223–238, Aug. 1996.

[4] A. Sinha, A. Wang, and A. Chandrakasan, "Algorithmic transforms for efficient energy scalable computation," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 2000, pp. 31–36.

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

[6] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler optimizations for low power systems," in *Power-Aware Computing*, R. Melhem and R. Graybill, Eds. Kluwer Academic Publishers, 2002.

[7] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The impact of source code transformations on software power and energy consumption," *J. Circuits, Systems, & Computers*, vol. 11, no. 5, pp. 477–502, May 2002.

[8] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Power analysis of embedded operating systems," in *Proc. Design Automation Conf.*, June 2000, pp. 312–315.

[9] 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.

[10] 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 Wkshp.*, Sept. 2000.

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

[12] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. ACM Symp. Operating Systems Principles*, Dec. 2001, pp. 89–102.

[13] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Task generation and compile-time scheduling for mixed data-control embedded software," in *Proc. Design Automation Conf.*, June 2000, pp. 489–494.

[14] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Synthesis of embedded software using free choice Petri nets," in *Proc. Design Automation Conf.*, June 1999, pp. 805–810.

[15] F. Thoen, J. Steen, G. De Jong, G. Goossens, and H. De Man, "Multi-thread graph: A system model for real-time embedded software synthesis," in *Proc. Euro. Design & Test Conf.*, Mar. 1997, pp. 476–481.

[16] A. Prayati, C. Wong, P. Marchal, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, H. De Man, and A. Birbas, "Task concurrency management experiment for power-efficient speed-up of embedded MPEG4 IM1 player," in *Proc. Int. Wkshp. Parallel Processing*, Aug. 2000, pp. 453–460.

[17] T. K. Tan, A. Raghunathan, and N. K. Jha, "Software architecture transformations: A new approach to low energy embedded software," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2003, pp. 1046–1051.

[18] E. Johansson and S.-O. Nystrom, "Profile-guided optimization across process boundaries," in *Proc. ACM SIGPLAN Wkshp. Dynamic & Adaptive Compilation Optimization*, Jan. 2000, pp. 23–31.

[19] E. Stenman and K. Sagonas, "On reducing interprocess communication overhead in concurrent programs," in *Proc. Erlang Wkshp.*, Nov. 2002.

[20] F. Catthoor, K. Danckaert, S. Wuytack, and N. D. Dutt, "Code transformations for data transfer and storage exploration preprocessing in multimedia processors," *IEEE Design & Test of Computers*, vol. 8, no. 3, pp. 70–82, Mar. 2001.

[21] POSIX Standard, IEEE Portable Application Standards Committee (http://www.pasc.org/).

[22] http://www.crhc.uiuc.edu/Impact.

[23] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, pp. 1–8, Dec. 1996.

[24] S. McPhail, "Development of a simple navigation system for the autosub autonomous underwater vehicle," in *Proc. Engineering in Harmony with Ocean*, Oct. 1993, pp. 504–509.

[25] 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.