Energy-Optimizing Source Code Transformations for Operating System-Driven Embedded Software

YUNSI FEI
University of Connecticut
SRIVATHS RAVI and ANAND RAGHUNATHAN
NEC Labs America
and
NIRAJ K. JHA
Princeton University

This paper proposes four types of source code transformations for operating system (OS)-driven embedded software programs to reduce their energy consumption. Their key features include spanning of process boundaries and minimization of the energy consumed in the execution of OS services—opportunities which are beyond the reach of conventional compiler optimizations and source code transformations. We have applied the proposed transformations to several multiprocess benchmark programs in the context of an embedded Linux OS running on an Intel StrongARM processor. They achieve up to 37.9% (23.8%, on average) energy reduction compared to highly compiler-optimized implementations.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Concurrency*; D.4.4 [**Operating Systems**]: Communication Management—*Buffering*; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms: Performance, Management

Additional Key Words and Phrases: Linux, energy consumption, source code transformations

ACM Reference Format:

Fei, Y., Ravi, S., Raghunathan, A., and Jha, N. K. 2007. Energy-optimizing source code transformations for operating system-driven embedded software. ACM Trans. Embedd. Comput. Syst. 7, 1, Article 2 (December 2007), 26 pages. DOI = 10.1145/1324969.1324971 http://doi.acm.org/10.1145/1324969.1324971

This work was supported by DARPA under contract no. DAAB07-02-C-P302.
Authors' addresses: Yunsi Fei, Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT 06269; email: yfei@engr.uconn.edu; Srivaths Ravi and Anand Raghunathan, NEC Labs Inc., Princeton, NJ 08540; email: {sravi;anand}@nec-labs.com; Niraj K. Jha, Department of Electrical Engineering, Princeton University, Princeton, NJ 08544; email: jha@princeton.edu.

1. INTRODUCTION

Limited battery life has made energy efficiency a critical issue for mobile computers and portable embedded systems, such as laptops, personal digital assistants (PDAs), cell phones, etc. Researchers have traditionally focused on reducing the energy consumption by redesigning the hardware components of embedded systems for low power [Raghunathan et al. 1998; Chandrakasan and Brodersen 1995; Rabaey and Pedram 1996], or utilizing software-controlled power management techniques that take advantage of custom low-power hardware [Benini and De Micheli 1997]. More recent studies have examined the architecture of the overall system for energy-saving opportunities [Benini and De Micheli 2000], which consider not only hardware components for energy reduction, but also energy-efficient software design and compilation.

Analogous to hardware power optimization techniques applicable to various levels in its design hierarchy, low-energy software design can also be performed at three levels of abstraction, which from bottom up are the instruction level, program or source-code level, and algorithm level [Chung et al. 2001].

Instruction-level techniques for low energy have been investigated extensively, and they are mainly employed during the compilation process [Tiwari et al. 1994; Su et al. 1994; Tiwari et al. 1996; Lee et al. 1997; Mehta et al. 1997]. For instance, these techniques 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 interinstruction overheads, etc. While these approaches can be automated in the compilation process, the overall energy consumption savings that can be achieved is of a smaller scale 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. Sinha et al. [2000] first introduced the notion of energy-scalable computation on general-purpose processors. Their algorithms demonstrate that by using simple transformations (with insignificant overhead) the energy-quality behavior of certain algorithms can be significantly improved. Simunic et al. [2000] considered alternative algorithms for implementing the same functionality, comparing algorithmic efficiency against the original one using a high-level energy estimator. The input data fidelity[1] of an algorithm can also be changed for saving energy [Flinn and Satyanarayanan 1999; Shenoy and Radkov 2003]. Narayanan and Satyanarayanan [2003] presented examples of computation fidelity alteration. Since these algorithmic approaches are mostly based on human intuition and knowledge, significant manual effort is needed for designing the alternative algorithms. 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. Several source code transformations

[1]*Fidelity* refers to the degree to which the scaled or transformed data resemble the original one.

proposed for reducing the software energy consumption have been surveyed in Kandemir et al. [2002] and Brandolese et al. [2002].

Traditional source code transformations do not consider the effects of the OS, and are only applicable to source code within a single process.[2] More importantly, they do not target coarse-grained system-level concurrency and global data flow among multiple processes. Our work is based on the fact that, in concurrent multiprocess programs, factors such as interprocess synchronization, data communication, context switches, OS intervention, etc., can significantly affect the overall energy consumption of an application. In this work, we propose a source code transformation methodology employing several techniques that span the OS and application boundaries to reduce the energy consumption of multiprocess OS-driven embedded software. We discuss how to systematically identify opportunities for the proposed transformations and apply them directly to the program source code.

1.1 Related Work

As embedded applications become increasingly sophisticated, an OS is utilized by many systems to manage both the hardware and software resources, and the software exercises the OS significantly. Thus, the effects of the OS on system performance and energy consumption need to be considered.

1.1.1 *OS-Related Energy Analysis and Optimization.* 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. [2003] first developed an energy profiler for applications executing on an embedded system based on the Fujitsu SPARCLite processor running a commercial real-time OS$-\mu$C/OS II. They demonstrated that the real-time OS consumes a significant fraction of system power and also impacts the power consumed by other software components. Tan et al. [2003a] developed an energy simulator called EMSIM for an embedded system featuring a StrongARM processor that uses embedded Linux as its OS. Vahdat et al. [2000] reexamined the design and implementation of OSs by taking energy consumption as the primary metric instead of performance. Lu et al. [2000] implemented an OS-directed power-management scheme in Linux and achieved significant power reductions compared to hardware-centric shutdown techniques. Pillai and Shin [2001] 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 while maintaining real-time guarantees. 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. Our work explores such opportunities.

[2]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.

1.1.2 *Software Synthesis of Task-Level Concurrency and Data Parallelism.* Software synthesis for embedded systems without 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. [2000] provided a procedure for extracting tasks from the processes with Petri nets and scheduling their execution efficiently on a single processor. This approach is useful in low-end embedded systems, where there is little OS support and, hence, software synthesis itself has to explicitly provide many OS services, such as scheduling. Sgroi et al. [1999] utilized a similar Petri net model and proposed a quasistatic scheduling algorithm for synthesizing a C code implementation from the functional specification. Thoen et al. [1997] presented a system-level multithread graph model to capture control, as well as data flow, concurrent behavior, interaction with the environment, and synchronization. Their previous work [Thoen et al. 1995] utilized a similar system specification and described an approach for mapping a multitasking system to a single processor under time constraints. Prayati et al. [2000] explored the issues of extracting concurrency between tasks in a system-level specification, performing scheduling in the presence of various constraints and mapping the tasks to a multi-processor system. They also proposed guidelines for improving the concurrency of the applications considered. Yu et al. [2004] presented a method for automatically generating embedded software from system specifications written in system-level design languages, e.g., SpecC and SystemC. Although they considered the issues of concurrency and communication with the support of a general real-time OS model, their work focuses on software synthesis (C code generation) and does not optimize interprocess communications specifically for energy consumption. Software architectural transformations were proposed as a means for reducing energy consumption in Tan et al. [2003b], based on optimizing an abstract software representation from which the program implementation is subsequently generated.

Other approaches that have dealt with optimization of concurrent processes include Johansson and Nystrom [2000] and Stenman and Sagonas [2002], which 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 and assume no OS support. Our work proposes several OS-driven source code transformations that reduce the energy overheads associated with process concurrency and OS services.

1.2 Paper Contributions

In this paper, we propose a source-level methodology employing a suite of source code transformation techniques to reduce the energy overheads associated with application/OS interactions as follows:

- Concurrent processes can be managed in an energy-efficient manner through merging or splitting of processes (*process-level concurrency management*).

- Interprocess 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*).

These optimization techniques work on program source code directly. Compared with software synthesis work, our proposed approach is a comprehensive end-to-end transformation process that does not depend on preoptimized system specifications or constraints. It is very flexible and can also be applied to optimize legacy code for energy savings.

We experimentally demonstrate the efficacy of these transformations in the context of several multiprocess 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.

The rest of this paper is organized as follows. Section 2 presents the model used in this work for multiprocess embedded software. Section 3 first describes the overall flow of source code transformations for energy optimization, and then demonstrates the effectiveness of the proposed source code transformation techniques using experimental results from several examples. Section 4 applies the flow of the proposed suite of transformations to two large case studies and analyzes their impact on energy reduction. Section 5 concludes the paper.

2. 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 multiprocess embedded systems and their specification. We then present an abstract 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.

2.1 Multiprocess Embedded Systems: An Overview

We assume a multiprocess embedded system is specified as a set of concurrently communicating sequential processes that is implemented on a specific single-processor platform. A process has no inner concurrency and is implemented as a sequential thread of execution. For each process, a set of input and output ports is defined, and point-to-point communication occurs between processes through channels between ports [Cortadella et al. 2000; Tan et al. 2003b]. The system communicates with the environment (for example, the disk, network adapter card, display console, UART, 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. The primary input ports are classified 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 disk for a file 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. Consider, for example, a web server, which listens to some network. Once client requests arrive at the ports, they are processed and the corresponding web pages are sent back to the requesting client. We also call these uncontrollable ports event-driven. At least one process is required for each uncontrollable input port to react to the event on that port. Since controllable ports do not impose any constraints on system behavior, we only consider uncontrollable ports as primary inputs.

2.2 The Control/Data Flow Process Network

We obtain a hierarchical software representation called *the control/data flow process network* for the multiprocess embedded system. This model, at the top level, captures a process-level view of the software, in which only essential control/data flow and dynamic constructs (e.g., semaphores, IPC) 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 interprocess optimizations.

Figure 1 shows the control/data flow process network for an example-embedded software program that consists of five processes P_0, $P_1, \ldots, P_4$. Each process is represented by an oval in this model, and the call graph 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., $channel_1$, $channel_2$, $channel_3$). 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 average data volume communicated for a single communication instance (*volume*) as well as the number of such communications (*#IPC*). For example, the pipe connecting *read* and *write* nodes in the call graphs of processes P_1 and P_2 corresponds to $channel_1$ with a data volume of 250 bytes per communication and 10 IPCs. These data are available for both inter- and intraprocess communications.

External hardware devices in the process network are shown as grey boxes ($source_1$, $source_2$, *sink*). They communicate with the processes through uncontrollable primary input or output ports. Control flow among processes is shown as dashed arcs and the arcs are labeled with their synchronization mechanism (*semaphore*).

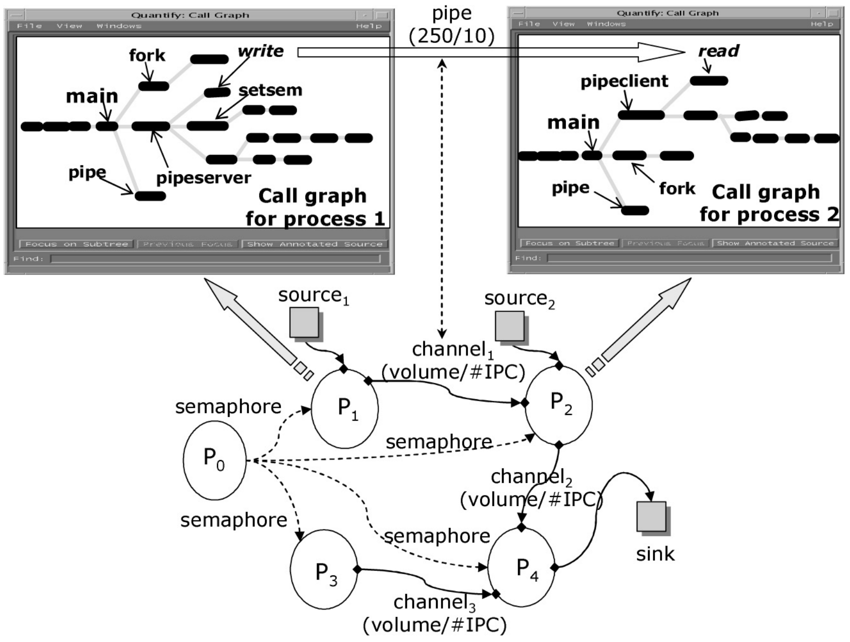
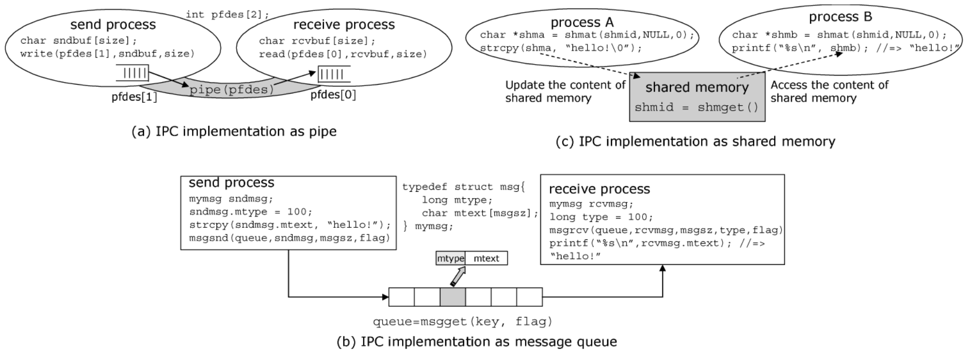


Fig. 1. A control/data flow process network.

2.3 Application Programming Interface (API) Description

In our work, we use an instruction-level energy simulator, EMSIM [Tan et al. 2003a], which targets embedded Linux on a single StrongARM processor, to assess the effect of the transformation techniques. We focus on software written in the C programming language, which is a nonconcurrent imperative language. Identifying process boundaries and IPC in the source code requires an understanding of the application–OS interface (API). We consider the POSIX interface [POSIX], 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. The POSIX interface provides several functions that perform IPC and synchronization (such as semaphores, pipes, shared memory, message passing, signals, sockets, etc.). IPC information can be extracted from the program source code by identifying and analyzing calls to these functions from the application software.

Since there are no generic communication primitives in the C language, we do not identify general IPC mechanisms. Instead, 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. Figure 2a illustrates how pipe is used for IPC. First, a set of two file descriptors is declared globally (*int pfdes[2]*). A pipe connecting these two file descriptors is then created by calling *pipe(pfdes)* so that *pfdes[1]* is opened for writing and *pfdes[0]* is opened for reading. When the *send* process writes data to *pfdes[1]*, the *receive* process can read data from *pfdes[0]*. The two file descriptors in this


send process
receive process
pfdes[1]
pipe(pfdes)
pfdes[0]
(a) IPC implementation as pipe
process A
process B
Update the content of shared memory
shared memory
Access the content of shared memory
(c) IPC implementation as shared memory
send process
receive process
mtype
mtext
(b) IPC implementation as message queue

```
int pfdes[2];
```

```
char sndbuf[size];
write(pfdes[1],sndbuf,size)
```

```
char rcvbuf[size];
read(pfdes[0],rcvbuf,size)
```

```
char *shma = shmat(shmid,NULL,0);
strcpy(shma, "hello!\0");
```

```
char *shmb = shmat(shmid,NULL,0);
printf("%s\n", shmb); //=> "hello!"
```

```
shmid = shmget()
```

```
mymsg sndmsg;
sndmsg.mtype = 100;
strcpy(sndmsg.mtext, "hello!");
msgsnd(queue,sndmsg,msgsz,flag)
```

```
typedef struct msg{
   long mtype;
   char mtext[msgsz];
} mymsg;
```

```
mymsg rcvmsg;
long type = 100;
msgrcv(queue,rcvmsg,msgsz,type,flag)
printf("%s\n",rcvmsg.mtext); //=>
"hello!"
```

```
queue=msgget(key, flag)
```

Fig. 2. Implementation of three IPC mechanisms.

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. For multiprocess software, we identify all the read and write system calls in each process and only those pairs that are connected through pipes between two processes are considered as candidate IPCs for optimization.

Instances of system calls that operate on the same message queue (generated by the system call *msgget()*), *msgsnd()*, and *msgrcv()*, define the communication pair for the message-passing mechanism. As shown in Figure 2b, 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. As shown in Figure 2c, processes A and B communicate through the memory region (corresponding to the shared memory identifier *shmid* returned by *shmget()*). The command *shmat ()* returns the start address of the shared memory segment so that *shma* and *shmb* point to the same memory region. The example program of Figure 2c would produce the desired results only if access of the shared memory by process B occurs after process A completes its update.

2.4 Control/Data Flow Process Network Extraction

To manipulate the program source code for coarse-grained process-level transformations, we first need to extract some information from the program source code regarding process-level concurrency and IPC, and then apply transformations based on the derived information in order to manage concurrency and data communication across process boundaries.

Starting from the 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

```
...
pid1 = fork();
if(pid1 == 0) { //child1
  Handle_GPS();
}
...
pid2= fork();
if(pid2 == 0) { //child2
  Estimate_depth();
}
...
```

Fig. 3. A programming template for multiprocess software.

network model for a multiprocess embedded system software can be automatically generated by modifying advanced compilers such as IMPACT [Chang et al. 1991] and SUIF2 [Hall et al. 1996]. Currently, this automation is not in place. We instead do a work-around to generate the process network very quickly. This task is simplified if the embedded program is written to adhere to some simple guidelines. For example, the identification of processes is greatly simplified if the functionality of each process spawned by the fork() system function is encapsulated within a corresponding C function. Figure 3 shows the programming template for a multiprocess software application, in which functions *Handle_GPS*() and *Estimate_depth*() each correspond to a created process. Following this structure allows us to explicitly map an OS-managed process to an OS-independent function call, and the process boundary is defined automatically by the range of the corresponding function. Note that this guideline is not overly restrictive, since a subprogram of arbitrary complexity can still be encapsulated within each function corresponding to a process.

The three different IPC mechanisms, as described in Section 2.3, can be identified by analyzing the corresponding calls to the OS functions from the application. The different communication channels (e.g., pipes, message queues, shared memory region) are identified. The read (e.g., *read* and *msgrcv* system calls) and write (e.g., *write* and *msgsnd* system calls) operations connected by one IPC channel form a communication pair. The corresponding ports are then extracted. In addition to the IPC mechanisms, any read and write operations that are not part of an IPC correspond to communication between a process and its environment (i.e., process communication from/to a *source*/*sink*, as shown in Figure 1). Thereby, the primary inputs/outputs, sources, and sinks are also identified. Synchronization between processes is often performed by certain system functions, such as semaphores, which are annotated in the compilation tool and identified. 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.

3. ENERGY MINIMIZATION THROUGH SOURCE CODE TRANSFORMATIONS

In this section, we describe a methodology for minimizing the energy consumption of OS-driven embedded software through source code transformations. Section 3.1 provides an overview of the entire flow, while Sections 3.2 and 3.3 give details of the important aspects.

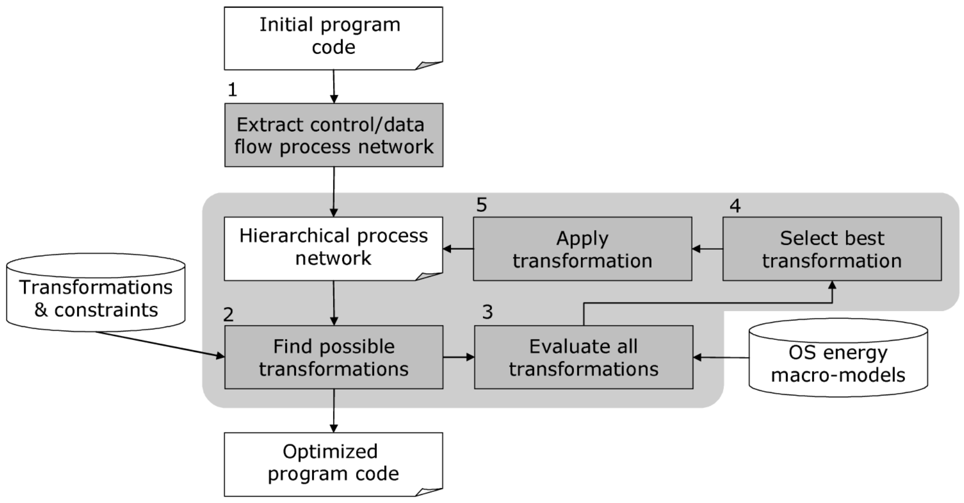

Initial program code
1
Extract control/data flow process network
5
Apply transformation
4
Select best transformation
Hierarchical process network
Transformations & constraints
2
Find possible transformations
3
Evaluate all transformations
OS energy macro-models
Optimized program code

Fig. 4. The source code level energy minimization methodology.

3.1 Overview of the Source Code Transformation Methodology for OS-Driven Embedded Software

We employ a design flow for systematically identifying the various transformations and applying them to the software. The proposed methodology is shown in Figure 4. In the figure, the white boxes represent the objects on which operations are performed; the grey boxes represent the operations in the algorithm. The methodology starts from the original C source code. We first profile it and extract the control/data flow process network from it (Step **1** in Figure 4), as described in Section 2. This acts as the initial process network. The energy consumption E_0 of the initial program can be obtained by running the instruction-level energy simulator—EMSIM [Tan et al. 2003a].[3] This high-level abstract view of the software enables us to identify and perform a sequence of source-level transformations. For a specific process network, we find all the possible transformations (Step **2**) under certain constraints, e.g., the number of active inputs. Currently, identification of transformation candidates is performed manually. These transformations transform the process network and are described in detail in Section 3.3. When the process network is described in a graph model, as shown in Figure 1, the interaction between processes (i.e., nodes in the graph), such as number of IPCs and average data volume of each IPC, etc., is annotated on the edges. This information helps identify possible transformation candidates. The methodology optimizes the software source code by applying a sequence of transformations, or moves, that maximizes energy savings. 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 evaluate all the possible transformations (Step **3**) and choose from them the one that yields most energy reduction (Step **4**). The best

[3]Note that the energy simulator needs to simulate the OS together with the application program, because the proposed source code transformations span the OS-application boundaries.

transformation is applied to the source code (Step **5**), and both the transformed program code and the new process network are obtained for the next iteration. When no further transformation is possible, the iterative process terminates with the energy-optimized source code as the output.

During the iterative process, for each source code transformation considered, we need to estimate the resulting change in energy consumption. Since this estimation is employed for a large number of candidates, it needs to be much more efficient than hardware or instruction-level simulation. The evaluation of the impact on energy consumption of each transformation is facilitated by several high-level energy macromodels, which include the IPC energy macromodels, as described in Section 3.3.4, as well as the energy macromodels for other OS services in the source code (obtained from Tan et al. [2005]).

After a sequence of source code transformations, the optimized program code is generated. We now use the low-level energy-simulation tool to obtain the energy consumption of the optimized source code and compare it with the original one for validation of energy reduction.

3.2 Energy Macromodeling at the Process Network Level

In the design flow for source code transformations, we can actually apply each candidate transformation to the source code obtained from the last iteration, generate the resultant program code, and utilize the low-level energy simulation framework to obtain the energy consumption. At each iteration, the inferior candidates can be pruned and the best transformation selected. The problem with this design space exploration approach is that it needs to perform a very slow low-level energy simulation for each transformation candidate. In addition, all the transformations need to be applied to the source code obtained from the last iteration. Efficient high-level software energy macromodels that can capture the change in energy consumption for each move are, hence, desirable.

All the possible transformations identified in Step **2** in Figure 4 target the abstract process network. This process-level view of software exposes global concurrency and communication among the coarse-grained processes, which actually induce the changes in energy consumption. We make the assumption that a source code transformation only changes the view of the process network, the computation performed in each process remains the same and the overall functionality of the software program does not change. Thus, energy macromodels need to be built for communications between processes and interactions between OS and application programs.

According to Tan et al. [2005], the OS energy characteristic data provide a number of energy macromodels that express the relationship between the energy consumption of software functions and some characteristic parameters. The energy macromodels can be classified into two categories:

- *The explicit set*: This includes the energy macromodels for those system functions that are explicitly invoked by application software, e.g., IPC mechanisms and other system calls. The energy macromodels express the dependency of the energy consumption of an atomic system function on some relevant parameters. For example, as described later in Section 3.3.4, the energy macromodel for a communication pair of a specific IPC mechanism in POSIX is given by:

$$E_{ipc}(x) = c_1 + c_2 x \tag{1}$$

where x is the number of bytes being transferred in each pair of calls and c_1 and c_2 are the coefficients of the macromodel. Suppose a source code transformation merges two processes. The amount of energy reduced by eliminating the IPCs between processes is given by:

$$\Delta E = c_1 N_{ipc} + c_2 x_{total} \tag{2}$$

where N_{ipc} is the number of times this IPC is invoked between the two specific application processes under consideration and x_{total} is the total number of communicated bytes. Values of parameters N_{ipc} and x_{total} are obtained by profiling the original software. Similarly, energy macromodels for other system functions, such as *fork()* and *malloc()*, are also derived in Tan et al. [2005].

- *The implicit set*: This includes the energy macromodels for those system functions that are not directly related to any OS primitive. Instead, they are the result of running the OS engine. The implicit functions include *timer interrupt*, *scheduling*, *context switch*, and *signal handling*. The context-switch function is of particular interest in evaluating the energy change because of process merging. The energy consumption of a one-time context switch is defined to be the amount of energy overhead incurred every time a process is switched out and another process is switched in. Since the context switch and other implicit OS functions do not involve data communication between processes, the energy macromodels for them represent the one-time energy overhead values. Statistics related to these implicit functions are also obtained by detailed profiling of the program source code using a low-level energy simulator.

The above energy macromodels were obtained for embedded Linux running on a StrongARM SA1110 microprocessor-based platform. The StrongARM SA1110 operates at a frequency of 206 MHz, operating voltage of 1.5 V, working temperature between 32° and 158°F, and is implemented in a 0.35 μm process technology. With the aid of high-level energy macromodels, the methodology presented in Figure 4 can select a sequence of transformations very quickly, apply them to the source code, and finally generate the energy-optimized program code output.

3.3 Transformation Techniques for Interprocess Optimization

In this section, we illustrate the various transformation techniques that explore the OS-application boundaries, and investigate their application to a few examples.

3.3.1 *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

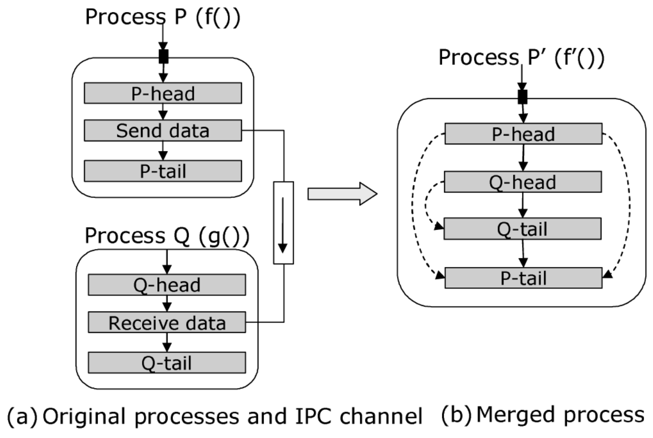


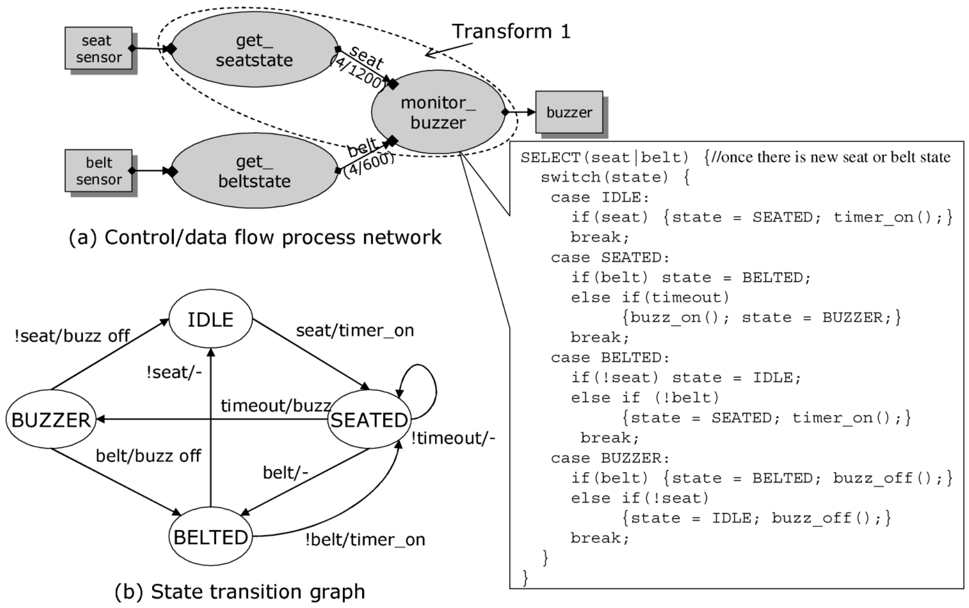
Fig. 5. The process merging operation.

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).

An example of concurrent process management is the process merging transformation shown in Figure 5. Send process P executes function $f()$ and receive process Q executes function $g()$, as shown in Figure 5a. Each function is divided into abstract blocks of code called *Head* (code preceding the communication operation), the communication operation *send*/*receive*, and *Tail* (the rest of the function). Process P is driven by an external active device, and a communication channel exists from the *send* in process P to the *receive* in process Q. When the two processes are merged, functions $f()$ and $g()$ get merged. In this example, we assume that a control dependency exists between the *Head* and *Tail* sections of each function and a define/use dependency between the variables in section *Head* of the sender function and section *Tail* of the receiver function. These dependencies are shown in Figure 5b as dashed lines in process P'. The implementation of process merging, therefore, requires a static coarse-grained scheduling step to keep these blocks of code in proper order. In addition, some local variables need to be renamed to avoid conflict after merging.

We now demonstrate 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 [Wolf 2001]. Figure 6a 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 Figure 6b, and the pseudocode 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 before the timer expires, a buzzer is turned on to alert the driver, and the



```
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. 6. An example of applying process merging transformation.

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 proper states automatically.

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 4 bytes. Thus, the two processes are candidates for merging (we refer to this transformation as *Transform* 1).
- Processes *get_beltstate* and *monitor_buzzer* communicate 600 times and each IPC corresponds to a data traffic of 4 bytes. Thus, these two processes 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 [Tan et al. 2003a] was used to generate the results. The energy data also show

Table I. Comparison between Original and Transformed Source Code for the Seat-Belt Example

Source Code	# Proc	# Channels	Total Energy (mJ)	Energy Reduction (%)
Original	3	2	24.27	–
Transform1 (merged with *get_seatstate*)	2	1	18.02	25.8
Transform2 (merged with *get_beltstate*)	2	1	19.62	19.2

that *Transform 1* is better than *Transform 2*. This is expected, since the overall interprocess data volume is significantly reduced in the former case compared to the latter (4 800 as opposed to 2 400 bytes).

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

- Two processes P and Q can be merged if 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.
- Two processes P and Q are good candidates for merging if the number of IPCs and the communicated data volume are high.
- For a single-processor platform, process merging is an effective way of reducing energy consumption incurred by IPC.

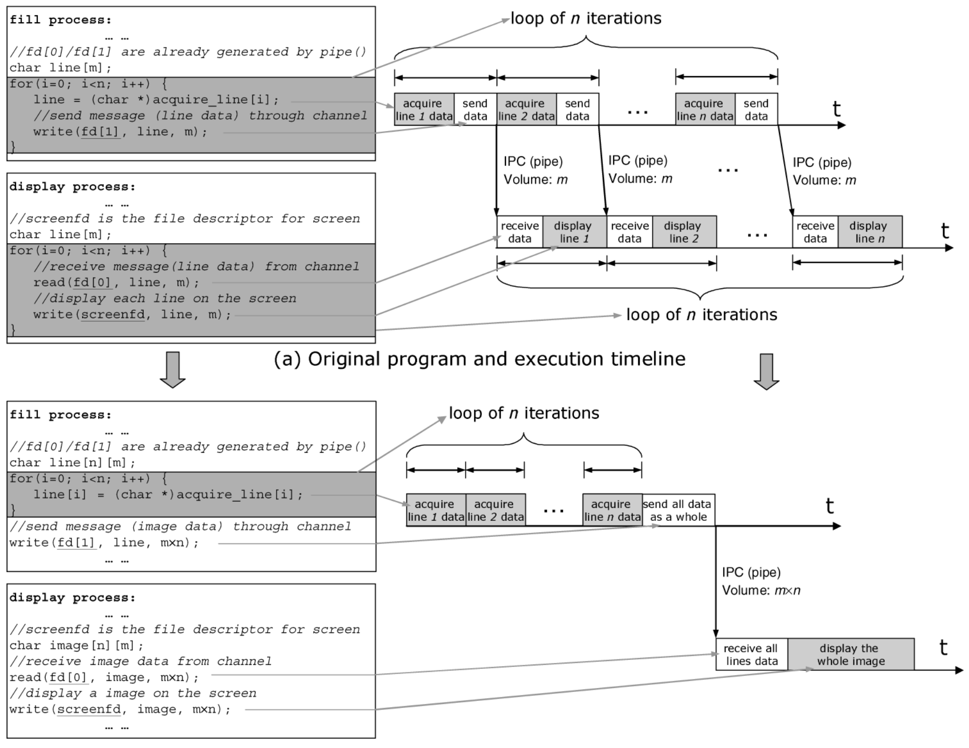
3.3.2 *Message Vectorization.* This transformation vectorizes the messages communicated between two processes, thereby reducing the number of IPCs. We now illustrate this transformation using the following example.

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 Figure 7a. 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.

Figure 7a 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 obtained by applying message vectorization to this application. Figure 7b 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 in the original program.



```
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. 7. Applying message vectorization to the *DrawLine* application.

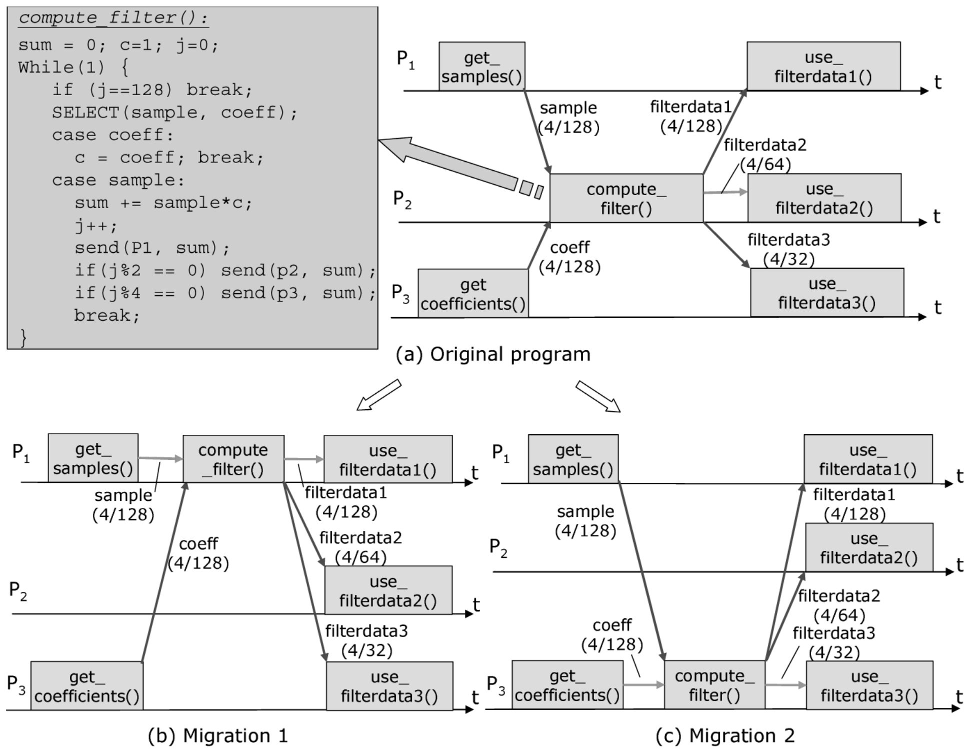
Table II. Comparison Between Original and Optimized Source Code for the DrawLine Application

Source Code	Total Energy (mJ)	# Proc	# Channels	# Send/Receive IPC Pairs
Original	8.24	2	1	1024
Optimized	7.17	2	1	1
Reduction	13.0%	0	0	1023

- The number of messages, as well as the number of sends and receives, is 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.

Table II summarizes the energy consumption results for the two cases. Because of message vectorization, the number of IPCs (# *send*/*receive pairs* in Table II) is reduced from 1024 to 1. The total energy consumption is reduced by 13.0%, thus demonstrating the efficacy of this transformation.



```
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. 8. An example of computation relocation among processes.

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

Example 3. Figure 8a 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 *filterdata*1, *filterdata*2, and *filterdata*3, which are used subsequently by functions *use_filterdata*1, *use_filterdata*2, and *use_filterdata*3.

Edges between different functions shown in the figure indicate the inter or intraprocess 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 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.

Table III. Inter- and Intraprocess 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 the computation corresponding to *compute_filter*() in process P_2. The function sums the data (*sample*) obtained from P_1 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 *filterdata*1 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. In other words, the number of communications is 64. Since *use_filterdata*2() and *compute_filter*() are in the same process, there is no IPC occurring, implying that the number of intraprocess communications is 64. Finally, results are filtered to P_3 at a quarter rate resulting in 32 IPCs from P_2 to P_3.

Assume that all the IPC mechanisms implemented in this example are the same. For any specific mechanism, consider the energy macromodel,[4] which is given by the equation.

$$E = \alpha + \beta \times x \tag{3}$$

In Eq. (3), 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 because of 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. Equation (3) can then be modified to yield the overall IPC cost for a process P_i and a computation C (not in P_i) as follows.

$$E_i = \alpha \times (n_s + n_r) + \beta \times (x_s + x_r) \tag{4}$$

For example, if we consider process P_1 and computation *compute_filter*(), then the various parameters in Eq. (4) are as shown Figure 8a. For process P_1, the total amount of source data communicated, x_s, is 4 × 128 bytes, and the number of source data communications, n_s, is 128. The amount of result data due to *compute_filter*() that is used by process P_1 is 4× 128 bytes (parameter x_r), and the number of result data communications is 128 (n_r). Table III lists the relevant information for the three processes.

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 macromodel given in Equation (4)

[4]IPC mechanisms and their energy macromodels are discussed in the next section.

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	ShM	Msg			
ORIGINAL	36.26	37.97	39.33	3	416	1664
MIGRATION1	**28.08**	**29.43**	**30.74**	3	224	896
MIGRATION2	30.99	31.17	32.33	3	320	1280
Optimized reduction	22.6%	22.5%	21.8%	0	192	768

for three different IPC mechanisms (pipe, message passing and shared memory). (The actual α and β values are provided later in Table V). From the results shown in Table IV, we can see that irrespective of the IPC mechanism, MIGRATION1 results in the most energy-efficient configuration.

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 $E_{ORIGINAL} = E_1 + E_3$. When computation is relocated to P_1 (MIGRATION1), as shown in Figure 8b, the IPC cost becomes $E_{MIGRATION1} = E_2 + E_3$. Similarly, when relocated to P_3, the IPC cost is $E_{MIGRATION2} = E_1 + E_2$. Thus, $E_{ORIGINAL} > E_{MIGRATION2} > E_{MIGRATION1}$, implying that relocating computation to P_1, as shown in Figure 8b, should be the most energy-efficient transformation. Table IV verifies the results by comparing the energy consumption obtained using EMSIM for the original and transformed source code. For each version of source code, the three different IPC mechanisms were implemented.

3.3.4 *IPC Mechanism Selection.* IPC mechanism selection refers to the process of choosing the most energy-efficient mechanism (from 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 illustrations of energy macromodels for three popular IPC mechanisms in embedded Linux.

In order to derive the energy macromodels, we created three test applications. Each application consists of a communication pair with one shared-memory, one message queue, and one pipe, respectively. For the shared-memory mechanism, this one-time IPC implies updating (write) and accessing (read) the shared-memory unit once. For message passing, it implies a pair of *msgsnd* and *msgrcv* system calls, while for pipes, it implies a pair of *read* and *write* pipe system calls. 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. We first obtained the energy estimates using the EMSIM simulator for each IPC mechanism by varying the number of IPC operations with a fixed data volume of each IPC. We then studied the dependency of IPC energy costs on the communication data

Table V. Energy Macromodels for Different IPC Mechanisms in Linux

IPC Mechanisms	Macromodels (nJ)
shared memory (without protection) (x bytes)	$E = 67.3 + 71.33x$
shared memory (with semaphore protection) (x bytes)	$E = 2294.5 + 71.33x$
pipe write/read (x bytes)	$E = 1892.6 + 2.45x$
msgsnd/msgrcv (x bytes)	$E = 2518.6 + 4.41x$

volume. This is because communication complexity (not just IPC frequency) affects performance and energy consumption.

Using the above energy characterization approach, we derived the energy macromodel for the three IPC implementations shown in Table V. The targeted platform features embedded Linux running on a StrongARM microprocessor, whose characteristics have been specified in Section 3.2. The macromodel consists of two terms. The first term corresponds to a base or constant cost, which reflects the energy overhead resulting from 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 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 implemented a semaphore-based synchronization method for shared memory; the second row of Table V shows the corresponding IPC energy macromodel. After considering the energy consumed by synchronization, we observed that shared-memory IPC is similar to the other two mechanisms in terms of energy consumption. We also observed that pipe tends to be more energy efficient compared to message passing. This is possibly due to the overheads associated with message copying and message queue handling (e.g., keeping the messages in the queue, type matching).

4. CASE STUDIES

This section presents the results of applying our framework with the proposed suite of source code transformations to two example software applications: an underwater navigation-control system [McPhail 1993] and an Ethernet packet-processing system [Dick et al. 2003]. 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 [Tan et al. 2003a] to estimate the energy consumption of an application (original and final 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.

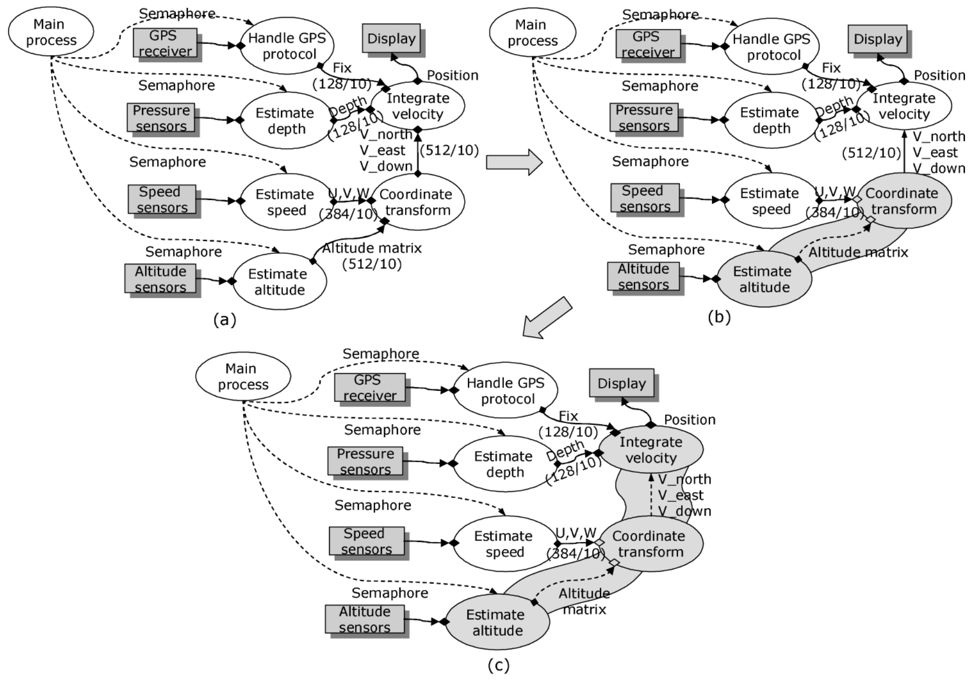


Fig. 9. The control/data flow process network for the underwater vehicle-navigation system and example process merging transformations.

We next present the experimental results and analysis for the two case studies in Sections 4.1 and 4.2, respectively.

4.1 Case Study: An Underwater Navigation System

We optimize the underwater vehicle navigation system described in McPhail [1993]. Figure 9a 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 Figure 9a with the IPC statistics generated through profiling. The original source code uses shared memory as its IPC mechanism.

We applied the transformations framework described in Section 3.1 to the original source code of the system and determined a sequence of transformations that can best reduce the energy consumption. These transformations are summarized in Table VI. The table lists the process-level statistics for the original and transformed source codes—number of processes (# proc) and

Table VI. Optimization Statistics for Original and Transformed Source Codes

Source Code	Trans	# Proc	# Channel	# IPC	IPC Volume (bytes)	Energy (mJ)
Original	—	7	5	50	16640	53.3
Ver2	PM	6	4	40	11520	44.1
Ver3	PM	5	3	30	6400	37.5
Ver4	MV	5	3	21	6400	35.9
Ver5	REPL	5	3	21	6400	34.7
Ver6	MV	5	3	12	6400	33.8
Ver7	MV	5	3	3	6400	33.1

communication channels (# channel), total number of IPCs (# IPC), and communicated data volume (IPC volume)—as well as the corresponding energy consumption (Energy). The transformations performed are as follows. Two process-merging transformations (PM) are performed first, as shown in Figure 9b and 9c. 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%). At this point in our design flow, the number of processes has been reduced to a minimum, i.e., the same number as the number of active devices.

The process-merging transformations are followed by three message vectorization (MV) instances interleaved with one IPC mechanism selection (REPL). These vectorizations correspond to the IPCs (*Fix, Depth*, and *U, V, W*) that are embedded in loops in the source code. The IPC mechanism selection (REPL) results in pipes replacing the shared-memory mechanism. The last two MV transformations are applied to the replaced new IPC mechanism of pipes. 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%.

Figure 10 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 suggest that considering process merging transformations first results in the best energy savings with the least number of transformations, since process merging indirectly influences the IPC (IPCs between processes are eliminated) and tends to have the largest impact on energy.

4.2 Case Study: Ethernet Packet-Processing System

The Ethernet packet-processing system [Dick et al. 2003] 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,

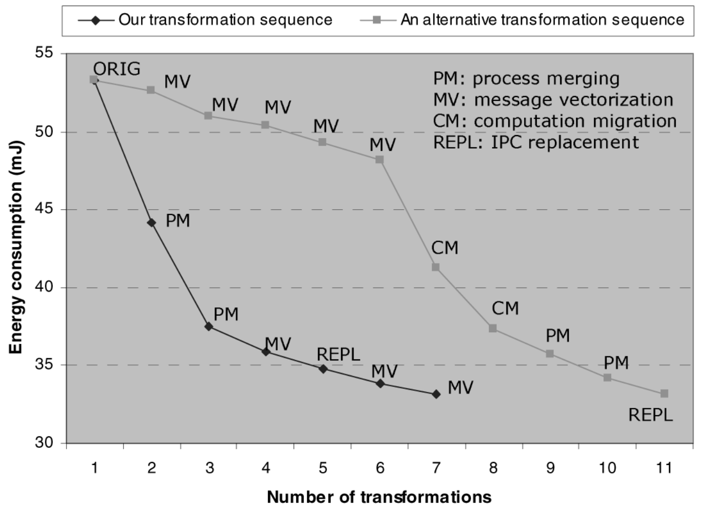


Fig. 10. Comparison of two different sequences of transformations.

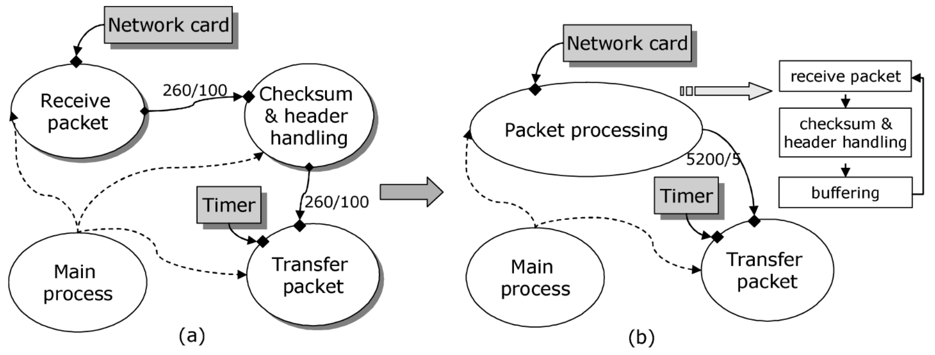


Fig. 11. The top-level control/data flow process network for the original and transformed implementations of an Ethernet packet-processing system.

and processes the packet header for transmission information. The packets are subsequently transmitted to the output devices periodically, driven by a timer.

Figure 11a 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.

Figure 11b 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

Table VII. Energy Consumption for Original and Transformed Source Codes

Source Code	#Proc	#Channel	#IPC	IPC Volume (bytes)	Energy (mJ)
Original	4	2	200	52,000	25.8
Process merging	3	1	100	26,000	21.2
Message vectorization	3	1	5	26,000	18.6

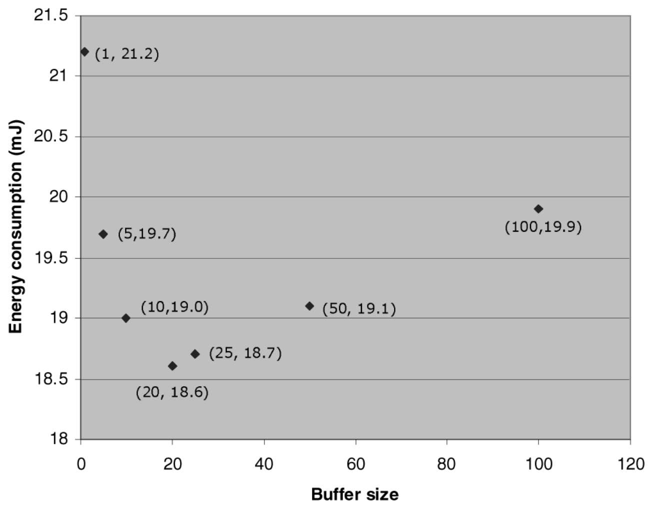


Fig. 12. Energy consumption variation with buffer size (after process merging).

listed in Table VII. 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 between the processes is reduced significantly. The IPCs are implemented with pipes in both the original and transformed implementations.

In this case study, we also analyzed the effect of buffer size on the overall energy consumption. Figure 12 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 can observe the trade-off between system memory usage (buffer size) and IPC overhead from this figure. Note that in Table VII, the buffer size selected for the "Message vectorization" transformation is 20 packets which will result in best energy savings.

5. CONCLUSIONS

In this work, we explored the OS-driven interface between processes in embedded software applications and proposed a set of source code-transformation techniques that reduce energy consumption. We manage process-level concurrency through process merging to save context-switch overheads and communications between processes. 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 process so as to reduce the number and data volume of IPCs. We believe that this set of transformations provides complementary optimization strategies to traditional compiler optimizations for energy savings. However, there is scope for further improvement, including extraction of the control/data flow process network through advanced compilation techniques, automation and improvement of the transformation methodology, etc.

REFERENCES

Benini, L. and De Micheli, G. 1997. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publ. Boston, MA.

Benini, L. and De Micheli, G. 2000. System-level power optimization techniques and tools. *ACM Trans. Design Automation Electronics Systems 5*, 2 (Apr.), 115–192.

Brandolese, C., Fornaciari, W., Salice, F., and Sciuto, D. 2002. The impact of source code transformations on software power and energy consumption. *J. Circuits, Systems, & Computers 11*, 5 (May), 477–502.

Chandrakasan, A. P. and Brodersen, R. W. 1995. *Low Power Digital CMOS Design*. Kluwer Academic Publ. Norwell, MA.

Chang, P. P., Mahlke, S. A., Chen, W. Y., Water, N. J., and Hwu, W. W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proc. Int. Symp. Computer Architecture*. 266–275.

Chung, E.-Y., Benini, L., and De Micheli, G. 2001. Source code transformation based on software cost analysis. In *Proc. Int. Symp. System Synthesis*. 153–158.

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

Dick, R. P., Lakshminarayana, G., Raghunathan, A., and Jha, N. K. 2003. Analysis of power dissipation in embedded systems using real-time operating systems. *IEEE Trans. Computer-Aided Design 22*, 5 (May), 615–627.

Flinn, J. and Satyanarayanan, M. 1999. Energy-aware adaptation for mobile applications. In *Proc. ACM Symp. Operating Systems Principles*. 48–63.

Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam, M. S. 1996. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer 29*, 12 (Dec.), 1–8.

Johansson, E. and Nystrom, S.-O. 2000. Profile-guided optimization across process boundaries. In *Proc. ACM SIGPLAN Wkshp. Dynamic & Adaptive Compilation Optimization*. 23–31.

Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. 2002. Compiler optimizations for low power systems. In *Power-Aware Computing*, R. Melhem and R. Graybill, Eds. Kluwer Academic Publ. Boston, MA.

Lee, M., Tiwari, V., Malik, S., and Fujita, M. 1997. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans. VLSI Systems 15*, 1 (Mar.), 123–135.

Lu, Y. H., Benini, L., and De Micheli, G. 2000. Operating-system directed power reduction. In *Proc. Int. Symp. Low Power Electronics & Design*. 37–42.

McPhail, S. 1993. Development of a simple navigation system for the autosub autonomous underwater vehicle. In *Proc. Engineering in Harmony with Ocean*. 504–509.

Mehta, R., Owens, R. M., Irwin, M. J., Chen, R., and Ghosh, D. 1997. Techniques for low energy software. In *Proc. Int. Symp. Low Power Electronics & Design*. 72–75.

Narayanan, D. and Satyanarayanan, M. 2003. Predictive resource management for wearable computing. In *Proc. Int. Conf. Mobile Systems, Applications, & Services*. 113–128.

Pillai, P. and Shin, K. G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. ACM Symp. Operating Systems Principles*. 89–102.

POSIX. Portable Operating System Interface — IEEE Portable Application Standards Committee. [`http://standards.ieee.org/catalog/olis/posix.html/`].

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

Rabaey, J. and Pedram, M., Eds. 1996. *Low Power Design Methodologies*. Kluwer Academic Publ. Norwell, MA.

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

Sgroi, M., Lavagno, L., Watanabe, Y., and Sangiovanni-Vincentelli, A. 1999. Synthesis of embedded software using free choice Petri nets. In *Proc. Design Automation Conf.* 805–810.

Shenoy, P. and Radkov, P. 2003. Proxy-assisted power-friendly streaming to mobile devices. In *Proc. SPIE/ACM Conf. Multimedia Computing & Networking*. 177–191.

Simunic, T., Benini, L., De Micheli, G., and Hans, M. 2000. Source code optimization and profiling of energy consumption in embedded systems. In *Proc. Int. Symp. System Synthesis*. 193–198.

Sinha, A., Wang, A., and Chandrakasan, A. 2000. Algorithmic transforms for efficient energy scalable computation. In *Proc. Int. Symp. Low Power Electronics & Design*. 31–36.

Stenman, E. and Sagonas, K. 2002. On reducing interprocess communication overhead in concurrent programs. In *Proc. ACM SIGPLAN Wkshp. on Erlang*. 58–63.

Su, C.-L., Tsui, C.-Y., and Despain, A. M. 1994. Low power architecture design and compilation techniques for high-performance processors. In *Proc. COMPCON*. 489–498.

Tan, T. K., Raghunathan, A., and Jha, N. K. 2003a. A simulation framework for energy consumption analysis of OS-driven embedded applications. *IEEE Trans. Computer-Aided Design 22*, 9 (Sept.), 1284–1294.

Tan, T. K., Raghunathan, A., and Jha, N. K. 2003b. Software architecture transformations: A new approach to low energy embedded software. In *Proc. Design Automation & Test Europe Conf.* 1046–1051.

Tan, T. K., Raghunathan, A., and Jha, N. K. 2005. Energy macromodeling of embedded operating systems. *ACM Trans. Embedded Computing Systems 4*, 1 (Feb.), 231–254.

Thoen, F., Cornero, M., Goossens, G., and De Man, H. 1995. Real-time multi-tasking in software synthesis for information processing systems. In *Proc. Int. Symp. System Synthesis*. 48–53.

Thoen, F., Steen, J., De Jong, G., Goossens, G., and De Man, H. 1997. Multi-thread graph: A system model for real-time embedded software synthesis. In *Proc. European Design & Test Conf.* 476–481.

Tiwari, V., Malik, S., and Wolfe, A. 1994. Power analysis of embedded software: A first step towards software power minimization. In *Proc. Int. Conf. Computer-Aided Design*. 384–390.

Tiwari, V., Malik, S., Wolfe, A., and Lee, T. C. 1996. Instruction level power analysis and optimization of software. *VLSI Signal Processing Systems 13*, 2(3) (Aug.), 223–238.

Vahdat, A., Lebeck, A., and Ellis, C. S. 2000. Every joule is precious: The case for revisiting operating system design for energy efficiency. In *Proc. ACM SIGOPS European Wkshp.* 31–36.

Wolf, W. 2001. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann, San Francisco, CA.

Yu, H., Domer, R., and Gajski, D. 2004. Embedded software generation from system level design languages. In *Proc. Asia & South-Pacific Design Automation Conf.* 463–468.

Received July 2004; revised February 2005; accepted September 2006