Input Space Adaptive Embedded Software Synthesis[*]

Weidong Wang
EE Department, Princeton University, Princeton, NJ
wwang@ee.princeton.edu

Anand Raghunathan
NEC USA, C&C Research Labs, Princeton, NJ
anand@ccrl.nj.nec.com

Ganesh Lakshminarayana
NEC USA, C&C Research Labs, Princeton, NJ
ganesh@ccrl.nj.nec.com

Niraj K. Jha
EE Department, Princeton University, Princeton, NJ
jha@ee.princeton.edu

Abstract

This paper presents a novel technique, called *input space adaptive software synthesis*, for the energy and performance optimization of embedded software. The proposed technique is based on the fact that the computational complexities of programs or sub-programs are often highly dependent on the values assumed by input and intermediate program variables during execution. This observation is exploited in the proposed software synthesis technique by augmenting the program with optimized versions of one or more sub-programs that are specialized to, and executed under, specific input sub-spaces.

We propose a methodology for input space adaptive software synthesis which consists of the following steps: control and value profiling of the input program, application of compiler transformations as a preprocessing step, identification of sub-programs and corresponding input sub-spaces that hold the highest potential for optimization, and transformation of the sub-programs to realize performance and energy savings. We have evaluated input space adaptive software synthesis by compiling the resulting optimized programs to two commercial embedded processors (*Fujitsu SPARClite*^{TM} and *Intel StrongARM*^{TM}). Our experiments indicate that our techniques can reduce energy consumption of the whole program by up to 7.8X (an average of 3.1X for *SPARClite* and 2.6X for *StrongARM*) while *simultaneously* improving performance by up to 8.5X (an average of 3.1X for *SPARClite* and 2.7X for *StrongARM*), leading to an improvement in the energy-delay product by up to 66.7X (an average of 8.2X for *SPARClite* and 6.3X for *StrongARM*), at the cost of minimal code size overheads (an average of 5.9%).

1 Introduction

The increase in embedded software content of portable low-power electronic systems and appliances has made it important to consider power dissipation issues during the design of embedded software. In this paper, we discuss a novel methodology and techniques for optimizing energy consumption and performance of embedded software through input space adaptive software synthesis. Our techniques can be applied in conjunction with traditional software compilation, and are processor independent. Programs optimized through input space adaptive software synthesis attain higher performance and energy efficiency by utilizing optimized versions of one or more sub-programs that are specialized to, and executed under, specific input sub-spaces.

1.1 Paper Overview and Contributions

Starting with an embedded software program to be optimized, and typical input traces that are used to profile the program and generate various statistics, we present techniques to perform the key steps involved in the optimization of embedded software, which consist of: (i) application of compiler transformations as a preprocessing step to better explore the potential of our technique, (ii) identification of sub-programs that hold the highest potential for optimization, (iii) selection of the input sub-space(s) whose occurrence can lead to significant reductions in the implementation complexity for the chosen sub-programs, and (iv) iterative application of the known compiler transformations to achieve performance and energy improvement.

We have evaluated the proposed technique for several embedded software functions and programs using energy/performance evaluation systems for *Fujitsu SPARClite* and *Intel StrongARM* processors, as well as through direct current measurement on the *Itsy* handheld computer [1]. Our experimental results demonstrate energy reductions of up to 7.8X, and *simultaneous* performance improvements of up to 8.5X. Further, we demonstrate that our techniques are robust with respect to variations in the input statistics.

The following key advantages of our technique (compared to traditional software compilation flows) are worth noting:

1. It is able to spot unique optimization opportunities that cannot be identified through an analysis of the software program alone (as a result, these opportunities cannot be exploited by conventional software compilation techniques). The proposed technique translates these unique optimization opportunities into significant improvements in energy and performance over and above those achieved by a traditional optimizing compiler.
2. It is processor independent. We achieve similar energy and execution time reductions when we apply our technique to two different commercial processors, as illustrated in Section 4.

1.2 Related Work

Previous work related to energy efficient software design can be broadly classified into the following categories: adaptation of conventional compilation techniques to target low energy, and exploiting power saving features present in embedded processors, such as shut down and dynamic clock/voltage scaling.

The use of instruction-level energy models to develop energy-driven compiler optimizations (including instruction reordering, reduction of memory operands, operand swapping in the Booth multiplier, efficient usage of memory bands, and a series of processor-specific optimizations) was proposed in [2]. In [3], a paradigm for hardware/compiler co-design is presented to minimize the activity in the memory hierarchy of a high-performance

[*]Acknowledgments: This work was supported by DARPA under contract no. DAAB07-00-C-L516.

processor. Source code optimization guidelines aimed at reducing energy consumption are presented in [4]. The idea of compressing the most commonly executed instructions so as to reduce the energy dissipated in the system memory hierarchy and buses is presented in [5, 6]. A compiler-assisted technique, based on a profile-driven code execution, which finds an optimal level of parallelism to trade off performance for energy efficiency, is proposed in [7]. The idea of pipeline gating is proposed in [8] for reducing energy consumption overheads of speculative execution in high-performance processors. The influence of high-level compiler optimizations on system power is investigated in [9], with a focus on loop and function transformations. The above energy optimization techniques do not exploit the basic principle underlying input space adaptive synthesis, hence they are complementary to, and can be used in conjunction with, the techniques proposed in this paper.

Statistical information about a program's inputs or variables can also be used for optimization. Properties of inputs or problem instances have been used in the design of algorithms that display improved average or amortized computational complexity or improved performance for specific parts of the input space [10]. Trace scheduling [11] exploits control-flow statistics by compacting frequently occurring sub-programs using code motion. Value profiling attempts to identify and exploit dynamic information about program values that cannot be derived using traditional static compile-time analysis [12, 13]. In parallel to our work, automatic source code specialization has been proposed in [14], in which selection of the functions to be optimized and the argument values is based purely on conventional profiling statistics, such as the frequency of occurrence, and the sub-programs to be optimized are restricted to function calls whose arguments frequently assume the same value.

Input space adaptive synthesis differs significantly from these techniques in the manner in which it exploits value statistics (*e.g.*, reducing computational complexity of sub-programs as opposed to deriving more compact VLIW schedules), the level of abstraction at which it is applied, and in the methodology and algorithms used (*e.g.*, the use of entropy based metrics to identify promising sub-programs and input sub-spaces).

The concept of input space adaptive design was first introduced in the context of hardware design in [15]. However, the idea of input space adaptive design was explored only in the domain of *high-level synthesis* of hardware ASICs. In our work, we have integrated the concept of input space adaptation into the *compiler flow of embedded software synthesis* and have used enabling compiler transformations to fully realize the impact on performance and energy in the context of software. The techniques developed in the context of hardware design do not apply in the context of embedded software compilation, necessitating a different methodology and algorithms.

2 Input Space Adaptive Software Synthesis

In general, we optimize the identified sub-program by translating the input sub-spaces into value constraints on variables, and iteratively applying known compiler transformations, which are not applicable in the context of the original program.

Example 1: Consider the program shown in Figure 1(a) that transforms a directed graph (DG) represented in the form of a connectivity matrix, which is frequently used in many algorithms [16]. The matrix $org[][]$ stores the initial graph. The input array $trans[][]$ and coefficient $coeff$ are used to transform the DG. The number of the nodes in DG is given by $SIZE$. When a given DG of size $SIZE$ is represented by a $SIZE \times SIZE$ matrix, the number of nonzero entries is equal to the number of edges (N_{edge}) in DG. For most practical graphs, $N_{edge} << SIZE^2$, which means that the matrix representing DG (*i.e.*, $org[][]$) is a sparse matrix. The computational complexity can be significantly reduced if we can take advantage of this fact. We demonstrate how to *automatically* perform such an optimization using input space adaptive software synthesis. Figure 1(a) shows how we can suitably transform the original program by adapting to the input statistics. Based on our analysis procedure, described in detail in Section 3, the sub-program, which includes $<< 1, ++4, *1, +1, +2$ within loop $L3$ in Figure 1(a) is selected as the target sub-program. The target sub-space identified by our procedure is described by equation $org[i1][i3] == 0$.

To qualify as an optimization target, the candidate behavior should exhibit significantly reduced complexity when the inputs belonging to the target sub-space are encountered and, the chosen input sub-space should occur with a high frequency. In this example, the input subspace described above occurs in the input trace with a frequency of 87.5% [1]. In the optimized program in Figure 1(a), the original sub-program, consisting of operations $<< 1, ++4, *1, +1, +2$, is optimized into the sub-program consisting of only $+3$. It is clear that the optimized sub-program would evaluate much faster and consume much less energy than the original sub-program. Therefore, the selected sub-program and input sub-space are valid for our purpose.

Figure 1(b) details how the chosen sub-program and input subspace are used to derive the optimized sub-program, by representing the sub-program as a data flow graph (DFG) and applying a sequence of well-known compiler transformations. It is important to note that, although the compiler transformations used in this procedure are known, they are applicable to the identified subprogram only in the context of the chosen input sub-space, *i.e.*, an optimizing compiler would not be able to apply these transformations by merely analyzing the original program. Statements in the selected sub-program are annotated with the names of the corresponding nodes in the DFG. For example, the statement $tmp1{+}{+}$ corresponds to node $++4$ in the DFG.

The inputs to the DFG in Figure 1(b) are $org[i1][i3]$ and $trans[i3][i2]$ and the outputs are $result1[i1][i2]$ and $result2[i1][i2]$. **Step 1** transforms the sub-programs by applying the input sub-space constraint, *i.e., org[i1][i3] == 0*, to the sub-program. By employing the *constant propagation* compiler transformation, the $<< 1$ operation in the DFG is eliminated in **Step 2**. Again, by applying *constant propagation* in **Step 3**, the $++4$ operation is replaced by constant value 1. At this stage, we observe that the $*1$ operation, $tmp2 = tmp1 \times trans[i3][i2]$, becomes $tmp2 = 1 \times trans[i3][i2] = trans[i3][i2]$. This enables us to eliminate the $*1$ operation by applying the *strength reduction* compiler transformation in **Step 4**. Now, we find that the $+1$ and $+2$ operation nodes in the DFG can be merged in **Step 5**, by using the *common-subexpression elimination* technique. Finally, we obtain our optimized sub-program, which includes only one operation node in the DFG. ∎

Based on the above example, the following aspects of our approach are worth noting:

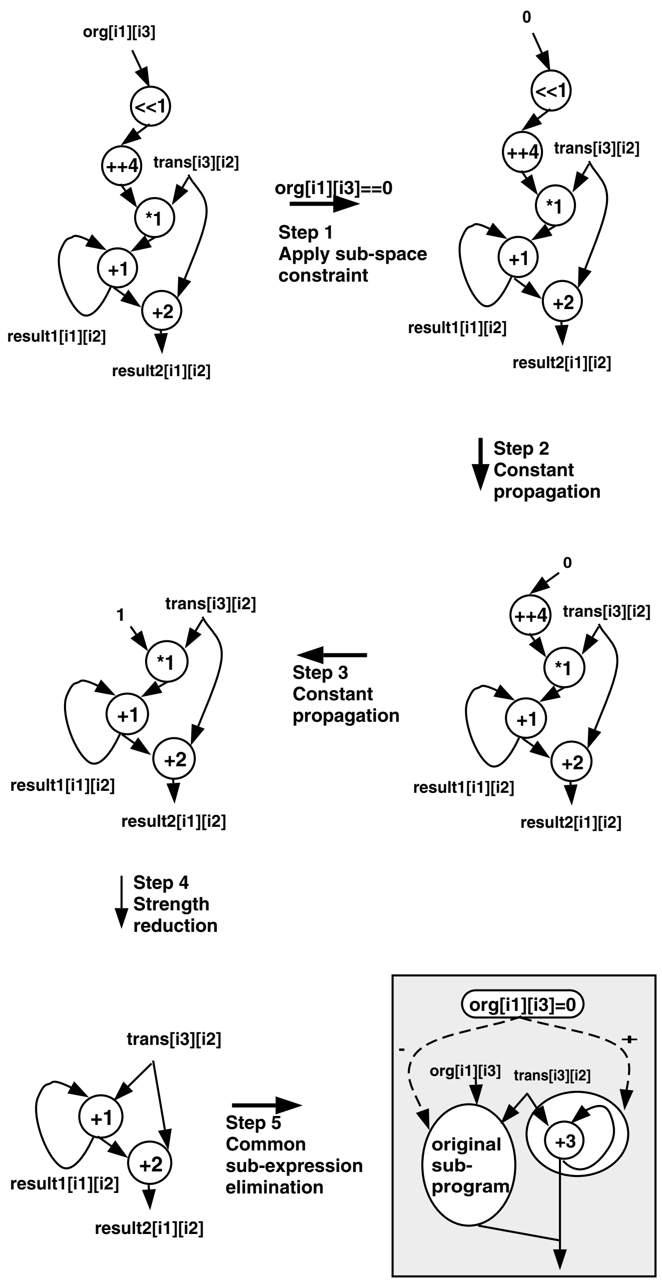
- The target input sub-space chosen for optimization should occur with a high frequency (and is, therefore, effective). In general, different application domains have distinct data

[1] Input sub-space frequency differs in different input traces. The energy/performance benefits of our technique are proportional to this frequency, rather than being based on the exact input trace itself. Hence, our technique is quite resilient with respect to variations in the input statistics, as demonstrated through experiments in Section 4.

```
Array dg(Array org, trans, int SIZE, coeff) {
   int tmp1, tmp2;
   Array result
   for(int i1 = 0; i1 < SIZE; i1++){              // <1, ++1(L1)
      for(int i2 = 0; i2 < SIZE; i2++) {          // <2, ++2(L2)
         result1[i1][i2] = result2[i1][i2] = 0;
         for(int i3 = 0; i3 < SIZE; i3++) {       // <3, ++3(L3)
            tmp1 = org[i1][i3] << 2;                   // <<1
            tmp1++;                                    // ++4
            tmp2 = tmp1 × trans[i3][i2];                //*1
            result2[i1][i2]=result1[i1][i2]+trans[i3][i2];//+2
            result1[i1][i2] +=tmp2;                     //+1
         }
      }
   }
   for(int i1 = 0; i1 < SIZE; i1++)               // <4, ++4(L4)
      for(int i2 = 0; i2 < SIZE; i2++)            // <5, ++5(L5)
         result1[i1][i2] = result1[i1][i2] × coeff;      // *2
   return result1, result2;
}
```

```
Array dg(Array org, trans, int SIZE, coeff) {
   int tmp1, tmp2;
   Array result
   for(int i1 = 0; i1 < SIZE; i1++){                 // <1, ++1(L1)
      for(int i2 = 0; i2 < SIZE; i2++) {             // <2, ++2(L2)
         result1[i1][i2] = result2[i1][i2] = 0;
         for(int i3 = 0; i3 < SIZE; i3++) {          // <3, ++3(L3)
            if(org[i1][i3] == 0){  //==1
               result1[i1][i2] += trans[i3][i2];             //+3
               result2[i1][i2] = result1[i1][i2]
            }
            else{
               tmp1 = org[i1][i3] << 2;                      // <<1
               tmp1++;                                       // ++4
               tmp2 = tmp1 × trans[i3][i2];                   //*1
               result2[i1][i2]=result1[i1][i2]+trans[i3][i2];//+2
               result1[i1][i2] += tmp2;                       //+1
            }
         }
      }
   }
   for(int i1 = 0; i1 < SIZE; i1++)                  // <4, ++4(L4)
      for(int i2 = 0; i2 < SIZE; i2++)               // <5, ++5(L5)
         result1[i1][i2] = result1[i1][i2] × coeff;          // *2
   return result1, result2;
}
```

(a)



(b)

Figure 1: The DG example: (a) Original and optimized programs in a high-level language, and (b) sequence of optimizations applied to the DFG of the chosen sub-program under the selected input sub-space

characteristics (*e.g.*, in the sparse matrix example, only a small fraction of the entries are non-zero values, *etc.*), and the optimization conditions chosen need to be identified appropriately. Most current optimization frameworks do not consider the program in the context of its application or domain data statistics, and would therefore be unable to spot such optimization opportunities.

- The energy consumption of input space adaptive programs depends on the choice of the sub-program and input sub-space. In the above example, the use of a different sub-space $org[i1][i3] == 1$ leads to a reduction from five operations to three operations, which is less beneficial than the reduction from five operations to one operation demonstrated in the example. This necessitates the development of accurate and efficient techniques to quantitatively assess the optimization potential for different input sub-spaces and sub-programs, as presented in Section 3.

```
Array dg(Array org, trans, int SIZE, coeff) {
  int tmp1, tmp2, tmp3;
  Array result
  for(int i1 = 0; i1 < SIZE; i1++){                    // <1, ++1(L1)
    for(int i2 = 0; i2 < SIZE; i2 += 2) {              // <2, +1(L2)
      i4 = i2 + 1;                                      // +2
      result1[i1][i2] = result1[i1][i4] = 0;
      result2[i1][i2] = result2[i1][i4] = 0;
      for(int i3 = 0; i3 < SIZE; i3++) {               // <3, ++3(L3)
        tmp1 = org[i1][i3] << 2;                        // <<1
        tmp1++;                                         // ++4
        tmp2 = tmp1 × trans[i3][i2];                    //*1
        tmp3 = tmp1 × trans[i3][i4];                    //*2
        result2[i1][i2]=result1[i1][i2]+trans[i3][i2];//+3
        result2[i1][i4]=result1[i1][i4]+trans[i3][i4];//+4
        result1[i1][i2] += tmp2;                        // +5
        result1[i1][i4] += tmp3;                        // +6
      }
    }
  }
  for(int i1 = 0; i1 < SIZE; i1++)                     // <4, ++4(L4)
    for(int i2 = 0; i2 < SIZE; i2++)                   // <5, ++5(L5)
      result1[i1][i2] = result1[i1][i2] × coeff;       // *2
  return result1, result2;
}
```

Figure 2: The DG example program after application of loop unrolling as an enabling step

While the above example demonstrates how the basic concept of input space adaptive software can be used to optimize energy and performance, its potential is not fully explored due to the fact that we have not optimized beyond the loop boundary. We compiled this example to the *Fujitsu SPARClite* and *Intel StrongARM* embedded processors and performed instruction-level energy estimation using the tools presented in [17] and [18]. Experimental results indicated energy reductions of 27.2% and 27.3%, respectively. In the next example, we demonstrate that targeted use of compiler transformations to preprocess the program can significantly increase the energy savings obtained through input space adaptive software synthesis.

Example 2: Consider again the original program shown in Figure 1(a). Having identified the target sub-program (indicated as the shaded block of code), we apply preprocessing compiler transformations to increase the potential for our technique, as explained next. In Figure 2, we unroll loop $L2$ by increasing the step length from 1 to 2 and reordering the instructions appropriately to enable our technique. After this step, we actually "merge" two of the previously selected sub-programs, which are in adjacent iterations of previous $L2$, into a new sub-program, which becomes the target of our technique. We also reduce some redundant operations. For example, $<< 1$ and $++4$ were previously executed for each iteration of the loop, but now we execute them only once, instead of twice, in the new loop iteration which is generated from two of the previous iterations. Note that the extent to which loop unrolling is performed presents a tradeoff, which can be evaluated using the quantitative techniques presented in Section 3. This step breaks the loop boundary and exposes more operations to our technique.

```
Array dg(Array org, trans, int SIZE, coeff) {
  int tmp1, tmp2, tmp3;
  Array result
  for(int i1 = 0; i1 < SIZE; i1++){                        // <1, ++1(L1)
    for(int i2 = 0; i2 < SIZE; i2 += 2) {                  // <2, +1(L2)
      i4 = i2 + 1; // +2
      result1[i1][i2] = result1[i1][i4] = 0;
      result2[i1][i2] = result2[i1][i4] = 0;
      for(int i3 = 0; i3 < SIZE; i3++) {                   // <3, ++3(L3)
        if(org[i1][i3] == 0){ // ==1
          result1[i1][i2] += trans[i3][i2];                 // +3
          result1[i1][i4] += trans[i3][i4];                 // +4
          result2[i1][i2] = result1[i1][i2]
          result2[i1][i4] = result1[i1][i4]
        }
        else{
          tmp1 = org[i1][i3] << 2;                          // <<1
          tmp1++;                                           // ++4
          tmp2 = tmp1 × trans[i3][i2];                      //*1
          tmp3 = tmp1 × trans[i3][i4];                      //*2
          result2[i1][i2]=result1[i1][i2]+trans[i3][i2];//+5
          result2[i1][i4]=result1[i1][i4]+trans[i3][i4];// +6
          result1[i1][i2] += tmp2;                          // +7
          result1[i1][i4] += tmp3;                          // +8
        }
      }
    }
  }
  for(int i1 = 0; i1 < SIZE; i1++)                         // <4, ++4(L4)
    for(int i2 = 0; i2 < SIZE; i2++)                       // <5, ++5(L5)
      result1[i1][i2] = result1[i1][i2] × coeff;           // *2
  return result1, result2;
}
```

Figure 3: The DG example: (a) after loop unrolling, and (b) after application of input space adaptive software synthesis to the unrolled program

Figure 3 shows the optimized program resulting from the application of input space adaptive software synthesis to the program in Figure 2. From this optimized program, we can see that under the input sub-space, the original sub-program, consisting of operations $<< 1, ++4, *1, *2, +3, +4, +5, +6$ in Figure 2, is replaced by the optimized sub-program which only contains operations $+3, +4$ in Figure 3. A larger complexity reduction is obtained, since we merged the two sub-programs which are under the same input sub-space, resulting in higher energy and performance improvements. When compiled to the *SPARClite* and *StrongARM* embedded processors, the optimized program shown in Figure 3 consumes, respectively, 44.9% and 39.5% less energy and executes in 45.2% and 40.2% less time, compared to the original program in Figure 1(a). After compilation using the *gcc* compiler, we compare the executable code size of the optimized program in Figure 3 and that of the original program. The result shows that the above improvement incurs only 3.4% increase in code size, which is quite small. ∎

From the above examples, the following observations can be made for optimizing programs with input space adaptive software synthesis:

- Our technique is input statistics driven, while most conventional software optimization techniques are input-independent. As experimental results will show, significant energy/performance benefits are still obtained even when the optimized program is subject to traces with very different input statistics.
- Sub-programs that account for a larger portion of the total execution time and energy consumption of the design are better targets for optimization.
- For a given sub-program, input sub-spaces that occur with a higher *probability* may yield larger savings.
- Different input sub-spaces lead to different reductions in the complexity of the chosen sub-program. The above two observations are incomplete since they do not consider the potential for complexity reduction. We capture the potential for simplification using an *entropy*-based metric that is presented in Section 3.

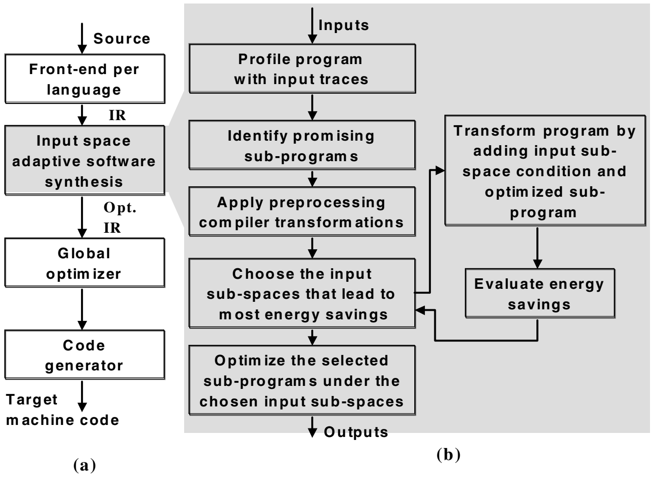
3 Methodology and Algorithms

In this section, we present an overview and algorithmic details for our software optimization technique. Section 3.1 presents the background in the context of a compiler flow and a brief description of input space adaptive software synthesis. Section 3.2 explains the important steps in detail.

3.1 An Overview

Figure 4(a) presents a compiler flow [19]. The shaded phase, *Input space adaptive software synthesis*, represents the phase where we apply our technique. Most of the high level and local optimizations occur in this phase, while in the lower phases, the optimizations mainly include detailed instruction selection, machine-dependent optimizations, *etc*. *IR* means intermediate representation and *Opt. IR* stands for optimized *IR*. The inputs to our algorithm are an intermediate representation of the program, typical input traces and designer-specified values for a set of optimization parameters. The major optimizations involved in this phase include procedure in-lining, loop transformations, algebraic transformation, constant propagation, strength reduction, common sub-expression elimination, or a sequence of compiler transformations. The output is the optimized input space adaptive program, which goes to the *Global optimizer* phase of the compiler flow.

Figure 4(b) shows the algorithm flow for input space adaptive software synthesis. We first profile the input program with the input traces. In addition to extracting control-flow statistics, we also compute entropy values for each variable in the program. For each input program, we identify the p most promising sub-programs as candidates for further optimization. Next, appropriate compiler transformations, *e.g.*, loop unrolling, procedure in-lining, *etc.*, are applied to the input program as preprocessing steps illustrated in Section 2. For each selected sub-program, we evaluate the energy effect of applying the input sub-space to the sub-program and select q input sub-spaces that lead to maximum energy reductions. Parameters p and q are user-supplied for



Source
Front-end per language
IR
Input space adaptive software synthesis
Opt. IR
Global optimizer
Code generator
Target machine code
(a)
Inputs
Profile program with input traces
Identify promising sub-programs
Apply preprocessing compiler transformations
Choose the input sub-spaces that lead to most energy savings
Optimize the selected sub-programs under the chosen input sub-spaces
Outputs
Transform program by adding input sub-space condition and optimized sub-program
Evaluate energy savings
(b)

Figure 4: The algorithm: (a) compiler flow, and (b) algorithm for input space adaptive software synthesis

controlling the aggressiveness of optimization. Finally, the chosen sub-programs are simplified by applying optimizing compiler transformations after constraining their input variables. Note that the optimized sub-programs only execute under the chosen input sub-spaces.

Similar to the technique in [15], we employ an entropy-based metric in selecting the sub-programs and input sub-spaces. The entropy, Ent, of a variable, which can take one of N values, is described by the following equation

$$Ent = -\Sigma_{i=1}^{N} p_i log p_i \qquad (1)$$

In this equation, p_i is the probability that the variable takes the ith value. A random variable that is distributed uniformly in the range $[0, 2^{n-1} - 1]$ has an entropy of n. A variable that can take on two values with a probability of 0.5 each has an entropy of 1, and a variable that has a constant value has an entropy of 0. The above results suggest that entropy correlates well with a variable's information content [20, 21]. Thus, for a given set of input vectors, a lower output entropy implies that the outputs have lower information content, and can hence be realized by a simpler (more energy-efficient) program. Note that the entropy values are calculated only once, during the simulation step. Further details of how we use entropy in our procedure are provided in Section 3.2.3.

3.2 Details

In this section, we give further details of the above-mentioned steps.

3.2.1 Energy Evaluation

We need to evaluate the energy savings in selecting the sub-programs and input sub-spaces. Tiwari et al. [2] use instruction-level energy models to estimate the energy cost of a program. For any given program, P, its overall energy cost, E_P, is given by Equation (2). The base cost, B_i of each instruction, i, weighted by the number of times it is executed, N_i, is summed up to give the base cost of the program. To this, the circuit state overhead, $O_{i,j}$, for each pair of consecutive instructions, (i, j), weighted by the number of times the pair is executed, $N_{i,j}$, is added. The energy contribution, E_k, of other inter-instruction effects, k, (stalls and cache misses) that would occur during the execution of the program, is finally added.

$$E_P = \Sigma_i (B_i \times N_i) + \Sigma_{i,j} (O_{i,j} \times N_{i,j}) + \Sigma_k E_k \qquad (2)$$

In a similar way, we can compute the energy cost of a program based on the energy evaluation of sub-programs, as opposed to individual instructions. Note that when large sub-programs, such as basic blocks, are considered, most of the inter-instruction effects in the second and third terms of Equation (2) have been accounted for in the base cost of sub-programs. Therefore, for a given program, P, we compute its energy cost, E_P, using Equation (3), where $E(\sigma_i)$ represents the base cost of sub-program σ_i, and $N(\sigma_i)$ represents the number of times that σ_i is executed.

$$E_P = \Sigma_i(E(\sigma_i) \times N(\sigma_i)) \tag{3}$$

If we replace a sub-program, σ, with an optimized sub-program, σ_{opt}, the optimized energy cost for σ we can achieve is shown in Equation (4). $\sigma_{org}, \sigma_{opt}$ represent the original and optimized sub-programs, respectively. $N(\sigma_{org}) + N(\sigma_{opt}) = N(\sigma)$ is the total number of times that σ is executed in the original program.

$$E(\sigma) = E(\sigma_{org}) \times N(\sigma_{org}) + E(\sigma_{opt}) \times N(\sigma_{opt}) \tag{4}$$

Equation (5) shows the energy savings we achieve by optimizing σ.

$$\begin{aligned} E_{saving} &= (E(\sigma_{org}) - E(\sigma_{opt})) \times N(\sigma_{opt}) && (5) \\ &= \delta_E \times \frac{N(\sigma_{opt})}{N(\sigma)} \times N(\sigma) && (6) \\ &= \delta_E \times p(\sigma) \times N(\sigma) && (7) \end{aligned}$$

From Equation (7), we observe that the energy saving is directly related to $p(\sigma)$, the probability that the optimized sub-program is executed, and the energy difference between the original and optimized sub-program, δ_E. In Section 3.2.3, we describe how to select the sub-programs and input sub-spaces based on Equation (7).

3.2.2 Compiler Transformations

In our algorithm, we employ compiler transformations in two steps: *by applying preprocessing compiler transformations*, and *transforming the program by adding input sub-space conditions and optimized sub-programs*. In Section 2, we illustrated in detail how we employed compiler transformations in these two steps. The computational complexity of determining an optimal sequence of transformations is known to be very high [19]. Hence, for each of the above two steps, we consider only a subset of all possible compiler transformations as candidates, based on an analysis of transformations that are most applicable and beneficial at each step.

In the preprocessing step, we found *loop transformations, including loop interchange, loop fusion, loop fission, and loop unrolling; procedure inlining; algebraic transformation, e.g., associativity, distributivity, etc.*, to be most beneficial. Therefore, we choose the above-mentioned compiler transformations as candidates in this step. In transforming the program by adding input sub-spaces, we feed the selected sub-program, along with the input sub-space, to procedures which perform the compiler optimizations under the input sub-space constraint [22]. The transformations used in this step include *constant propagation, strength reduction, and algebraic transformations*.

3.2.3 Sub-programs and Input Sub-spaces

The concept of sub-program and input sub-space was first introduced in [15]. We will revisit this problem in the context of software synthesis. From Equation (7), we see that the energy saving is directly related to the energy difference of the original and optimized sub-programs. Unfortunately, we do not have the exact energy consumption number of the optimized sub-program without knowing the input sub-spaces. What we can do is to define a metric to evaluate the optimization potential of the sub-program, as discussed next.

In Section 3.1, we claimed that, for a given set of input vectors, a lower output entropy value implies that the outputs have lower information content [20, 21], and can hence be realized by a simpler (more energy-efficient) program. We calculate the *word-level* entropy. This step is performed only once during the simulation step. The average entropy value of the outputs of σ is computed using Equation (8), where M is the number of the word-level outputs, O_i, belonging to σ.

$$avg\_entropy(O) = Q_\sigma(O) = \frac{1}{M}\Sigma_{i=1}^{M} Ent(O_i) \tag{8}$$

In cases where the output, O, of a sub-program is optimized to be proportional to one of its inputs, I, $Ent(\frac{O}{I}) = 0$ since $\frac{O}{I}$ is a constant, while $Ent(O) = Ent(I)$ might be high. Obviously, this is a good candidate for our technique. Therefore, we also take into account the relative entropy, *i.e.*, the entropy of the ratio of the word-level values of the output and input. A sub-program with more instructions clearly has more potential to be optimized by our technique. We use instr_count, $C(\sigma)$, to represent the number of instructions executed when σ is encountered. Equation (9) shows the metric we use to select the sub-program. The larger the value of $metric1(\sigma)$, the higher the potential of σ for input space adaptive software synthesis.

$$metric1(\sigma) = \frac{C(\sigma)}{min(Q_\sigma(O), Q_\sigma(\frac{O}{I}))} \tag{9}$$

In selecting the sub-programs to be optimized, a *bottom-up* method is employed, which means that each sub-program is initially formed from a single instruction. Each time we group a neighboring instruction, we evaluate *metric1*, until the value of *metric1* is maximized. The sub-program generated is saved for later optimization. We then start with an instruction outside the generated sub-programs and repeat the above process until all the instructions are in one of the sub-programs, and p sub-programs with the largest *metric1* are chosen, where p is a user-provided parameter to control the aggressiveness of optimization.

The next step is to identify q input sub-spaces for each selected sub-program. In Equation (7), we observe that the probability that the input sub-space is executed directly affects the energy savings. We define another metric, *metric2*, for selecting the input sub-spaces:

$$metric2(\sigma, \rho) = metric1(\sigma, \rho) \times p(\sigma, \rho) \tag{10}$$

where $metric1(\sigma, \rho)$ represents the *metric1* value for sub-program σ under input sub-space ρ and $p(\sigma, \rho)$ represents the probability that σ is executed under sub-space ρ.

It is computationally too expensive to try all the possible input sub-spaces. In our algorithm, the input conditions (input sub-spaces, in our terminology) we employ include $=, <, >, \neq, \geq, \leq$. The optimization conditions we use include: (i) one-term: $I_i = 0, 2^t; t = 0, 1, 2, \ldots$, where I_i is one of the S inputs of the sub-program, and (ii) two-term: $I_i = (\neq, <, >, \geq, \leq) I_j; i, j = 1, 2, \ldots, S$, where I_i and I_j are two of the S inputs of the sub-program. For each sub-program, we select q conditions (one-term or two-term) that result in the largest *metric2* value.

4 Experimental Results

We applied our technique to several embedded software programs. Typical input traces were assumed to be available for all the programs. Note that it is only the input statistics of the input

Table 1: *SPARClite* results: delay(10^{-3}s), energy(10^{-3}J) and energy×delay(10^{-6}J×s)

program	*original*			*optimized*						
	energy	*delay*	*E*D*	*energy*	*E.S.(%)*	*delay*	*P.I.(%)*	*E*D*	*E*D.R. (%)*	*C.O. (%)*
DG	104.2	113.3	11805.9	57.4	44.9	62.1	45.2	3564.5	69.8	3.4
LP	128.1	139.6	17882.8	37.9	70.4	40.9	70.7	1550.1	91.3	20.9
MUL	5.3	5.7	30.2	0.7	86.8	0.8	86.0	0.6	98.2	1.3
DFT	26.0	28.0	728.0	3.5	86.5	3.9	86.1	13.7	98.1	0.8
FINITE	21.2	23.3	494.0	9.1	57.1	9.9	57.5	90.1	81.8	8.5
COSINE	15.6	16.9	263.4	3.8	75.6	4.2	75.2	16.0	94.0	8.6
GCD	70.1	77.2	5411.7	30.0	57.3	35.6	53.9	1068.0	80.3	1.0
CHKSUM	28.2	31.0	874.2	9.6	66.0	10.5	66.1	100.8	88.5	2.8

Table 2: *StrongARM* results: delay(10^{-6}s), energy(10^{-6}J) and energy×delay(10^{-9}J×s)

program	*original*			*optimized*					
	energy	*delay*	*E*D*	*energy*	*E.S.(%)*	*delay*	*P.I.(%)*	*E*D*	*E*D.R. (%)*
DG	648.8	1799.0	1167.2	392.5	39.5	1076.0	40.2	422.3	63.8
LP	539.1	1491.0	803.8	203.1	62.3	558.0	62.6	113.3	85.9
MUL	61.9	160.1	9.9	27.3	55.9	72.1	55.0	2.0	80.2
DFT	232.7	673.0	156.6	30.1	87.1	79.0	88.3	2.4	98.5
FINITE	788.0	2020.4	1592.1	353.2	55.2	912.0	54.9	322.1	80.0
COSINE	130.9	337.0	44.1	54.8	58.1	139.2	58.8	7.6	82.7
GCD	900.7	2422	2181.5	316.3	64.9	854	64.7	270.1	87.6
CHKSUM	183.1	490.2	89.8	47.0	74.3	125.1	74.5	5.9	93.5

traces that are important, not the traces themselves. The original programs were optimized by applying the procedure described in Section 3. We compiled the programs to *Fujitsu SPARClite* and *Intel StrongARM SA-1100* embedded processors and performed instruction-level energy estimation [17, 18] to evaluate the energy savings. The accuracy of these energy estimation techniques has been confirmed earlier (*e.g.*, an accuracy of 97% in [18]). The resulting programs were compared with respect to the following metrics: *performance*, *energy*, *energy-delay product* and *code size*. The results obtained are summarized in Tables 1 and 2. Of our benchmarks, DG, which implements the transformation of Directed Graph, was discussed in Section 2. LP represents a linear predictor program. MUL performs multiplication of two polynomials. Discrete Fourier transform (DFT), finite impulse response (FINITE) filter and discrete cosine transform (COSINE) are well-known signal processing benchmarks [23]. GCD represents the program which computes the greatest common divisor of two integers. CHKSUM stands for the widely used checksum program.

In Tables 1 and 2, major column *program* represents the name of the software programs. Columns *energy, delay* and *E*D* represent energy consumption, execution time, and energy-delay product, respectively. Columns *E.S.*, *P.I.* and *E*D.R.* represent the energy savings, performance improvement and energy-delay product reduction, respectively.

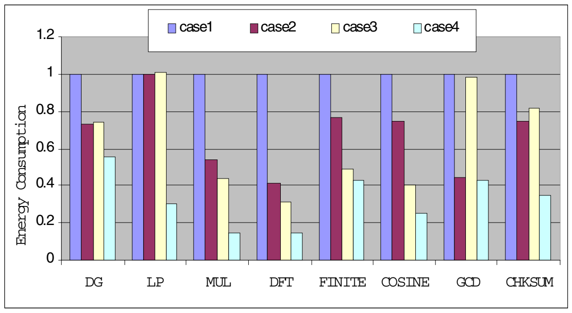
By employing our technique, the average energy reductions for *SPARClite* and *StrongARM* processors are 68.1% (*i.e.*, 3.1X) and 62.1% (*i.e.*, 2.6X), respectively (the averages were calculated based on comparing the sum of the values in the respective columns). Note that our techniques result in *simultaneous* energy savings and performance improvements. Hence, we also present performance results for our technique. The average performance improvements for *SPARClite* and *StrongARM* processors are 67.6% (*i.e.*, 3.1X) and 62.4% (*i.e.*, 2.7X), respectively. Note that energy consumption savings closely follow improvements in performance. This is because the variation in power consumption of different instructions in the instruction set is typically small [18].

The $energy \times delay$ results for the programs (product of the *energy* column and *delay* column from Tables 1 and 2) are calculated and shown in columns *E*D* and *E*D.R.*. The product is reduced by an average of 87.8% (*i.e.*, 8.2X) for the *SPARClite* processor, and 84.1% (*i.e.*, 6.3X) for the *StrongARM* processor.

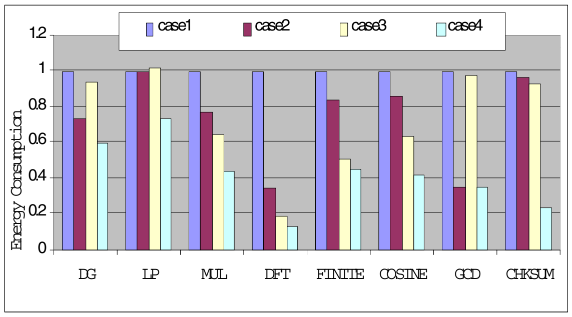
The energy results are also shown in Figure 5, in which the energy consumption of the original case is normalized to 1 for all examples. *Cases1-4* represent, respectively: *case1*: the original program, which corresponds to the *original* columns in Tables 1 and 2; *case2*: the optimized program without the compiler preprocessing step; *case3*: the original program after compiler optimization (preprocessing step); and *case4*: the optimized program with the compiler preprocessing step, which corresponds to the *optimized* columns in Tables 1 and 2. For all the examples, *case4* leads to the best program in terms of energy and performance.

Measurements on *Itsy* pocket computer: To verify the validity of our experimental results, we measured the energy consumption for the original and optimized versions of LP, FINITE, and CHKSUM by running them on the *Itsy v2.2* pocket computer [1]. The Itsy features a *StrongARM SA-1100* processor running at 206 MHz, under the embedded Linux Operating System. The *National Instruments 6035E* Data AcQuisition (DAQ) system was used to sample the current consumption of the processor, and the samples were fed into the *LabView* software and analyzed to compute the energy consumption for the time window of interest. The measured energy savings resulting from input space adaptive software synthesis were as follows: 66.2% for LP, 54.1% for FINITE, and 80.2% for CHKSUM. These correspond closely to the *E.S.* percentages in Table 2.

Code size overhead: To see the overhead of our technique, we also compiled all the examples with the *gcc* compiler, and compared the executable code size of the optimized program and that of the original one. The result is shown in column *C.O.* in Table 1 only, because we observed the overhead to be processor-independent and compiler-dependent. The overall increase in code size is very small (average of 5.9%).



(a)



(b)

Figure 5: Energy results for input space adaptive software synthesis: (a) *SPARClite* processor, (b) *StrongARM* processor

Sensitivity to input statistics: As explained in Section 2, our technique is trace driven. However, as mentioned earlier, the extent of energy and performance improvements are dependent upon the characteristics of the input trace, rather than the exact input trace itself. Specifically, the probability with which the chosen input sub-space occurs is the primary factor in determining the improvements. In order to study this dependency, we compiled the original and optimized programs for the DG and LP examples to *Fujitsu SPARClite* processor, and performed instruction-level energy estimation to evaluate the energy savings under different input traces. For the DG example (*case2 vs. case1*), when the input sub-space occurred with probabilities of 87.5% (as in Example 1), 50%, and 10.9%, the energy reductions were 27.2%, 17.3%, and 6.9%, respectively. For the LP example (*case4 vs. case1*), under sub-space probabilities of 85.7%, 42.9%, and 14.3%, the energy reductions were 70.4%, 44.3%, and 24.6%, respectively. Similar results were obtained in the case of the *Intel StrongARM* processor, and for the other example programs we considered.

From the above experiment, we can conclude that the energy and performance improvements are quite robust to variations in the input trace or input statistics. Hence, we believe that input space adaptive software synthesis can be quite useful even if only partially accurate information is available about the operating environment of the program (*i.e.*, its typical input traces).

5 Conclusions

In this paper, we presented an input space adaptive software optimization technique for improving performance and energy consumption. We presented algorithms to perform the different optimization steps, including application of compiler transformations as a preprocessing step, selection of the sub-programs to be optimized and the targeted input sub-spaces, and transformations of the embedded software by optimizing the sub-programs. Experimental results demonstrated that our techniques produce optimized programs which perform significantly faster and, *simultaneously*, consume significantly less energy than the original programs, leading to over an order-of-magnitude improvement in the energy-delay product.

References

[1] “The Compaq Itsy Pocket Computer Project. (http://www.research.compaq.com/wrl/projects/itsy/index.html).”.

[2] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: A first step towards software power minimization,” *IEEE Trans. VLSI Systems*, vol. 2, pp. 437–445, Dec. 1994.

[3] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, “Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors,” in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 70–75, Aug. 1998.

[4] T. Simunic, G. De Micheli, and L. Benini, “Energy-efficient design of battery-powered embedded systems,” in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 212–217, Aug. 1999.

[5] E. Macii, A. Macii, M. Poncino, and L. Benini, “Selective instruction compression for memory energy reduction in embedded systems,” in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 206–211, Aug. 1999.

[6] H. Lekatsas, J. Henkel, and W. Wolf, “Code compression for low power embedded system design,” in *Proc. Design Automation Conf.*, pp. 294–299, June 2000.

[7] D. Marculescu, “Profile-driven code execution for low power dissipation,” in *Proc. Int. Symp. Low Power Electronics and Design*, pp. 253–255, Aug. 2000.

[8] S. Manne, D. Grunwald, and A. Klauser, “Pipeline gating: Speculation control for energy reduction,” in *Proc. Int. Symp. Computer Architecture*, pp. 1–10, June 1998.

[9] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, “Influence of compiler optimization on system power,” in *Proc. Design Automation Conf.*, pp. 304–307, June 2000.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. McGraw Hill, New York, 1990.

[11] J. A. Fisher, “Trace scheduling: A technique for global microcode compaction,” *IEEE Trans. Computers*, vol. 30, pp. 478–490, July 1981.

[12] B. Calder, P. Feller, and A. Eustace, “Value profiling and optimization,” *J. Instruction-level Parallelism (http://www.jilp.org/)*, vol. 1, pp. 1–6, Mar. 1999.

[13] R. Muth, S. Watterson, and S. Debray, “Code specialization based on value profiles,” in *Proc. Int. Static Analysis Symposium*, pp. 340–359, June 2000.

[14] E. Chung and L. Benini, “Automatic source code specialization for energy reduction,” in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 2001.

[15] W. Wang, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, “Input space adaptive design: A high-level methodology for energy and performance optimization,” in *Proc. Design Automation Conf.*, pp. 738–743, June 2001.

[16] R. Sedgewick, *Algorithms in C++*. Addison-Wesley, Reading, MA, 1992.

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

[18] A. Sinha and A. Chandrakasan, “JouleTrack - A web based tool for software energy profiling,” in *Proc. Design Automation Conf.*, pp. 220–225, June 2001.

[19] A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1985.

[20] D. Marculescu, R. Marculescu, and M. Pedram, “Information theoretic measures for energy consumption at the register-transfer level,” in *Proc. Int. Symp. Low Power Design*, pp. 81–86, Apr. 1995.

[21] F. N. Najm, “Towards a high-level power estimation capability,” in *Proc. Int. Symp. Low Power Design*, pp. 87–92, Apr. 1995.

[22] G. Lakshminarayana and N. K. Jha, “FACT: A framework for applying throughput and power optimizing transformations to control-flow intensive behavioral descriptions,” *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 1577–1594, Nov. 1999.

[23] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1975.