Profiling Driven Computation Reuse: An Embedded Software Synthesis Technique for Energy and Performance Optimization*

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

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

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

Abstract

It has been observed that even highly optimized software programs perform "redundant" computations during their execution, due to the nature (statistics) of the values assumed by input or internal program variables. For embedded software running on battery-powered systems, such computations can be viewed as unnecessary energy overheads, and hence represent opportunities for improvement in energy efficiency. We present a systematic methodology to identify and eliminate redundancies in the computations performed by embedded software programs, by exploiting opportunities that dynamically arise for *computation reuse*. We report the results of experiments on two different embedded systems — a detailed simulation model of a *Fujitsu SPARClite* based embedded system, and actual current measurements on a *Compaq iPAQ* PDA. Our results demonstrate that the proposed technique can reduce energy by up to 46.9% (average of 21.2% and 13.9% for the *SPARClite* based system and the *iPAQ*, respectively) while *simultaneously* improving performance by up to 45.8% (average of 20.7% and 16.8% for the *SPARClite* based system and the *iPAQ*, respectively), compared to well-optimized programs that do not employ such a technique.

1 Introduction

Studies in various fields of research indicate that input data for computations possess interesting properties, such as locality, repetitiveness, and redundancy. For example, Zipf's law [1], which has been shown to be applicable to web servers, documents, web caches, *etc.* [2], indicates that a few input vectors can account for most of the vectors fed to the application (*e.g.*, the top 5% of input vectors in terms of frequency of occurrence may account for 50% of all the input vectors).

The focus of this work is on computation reuse, which is one approach to minimizing computational redundancies in software programs. In our work, we use the term *computation reuse* to refer to the reuse of results from specific execution instances of a sub-program to avoid executing future instances of the same sub-program. This paper presents a systematic computation reuse methodology for embedded software, with an objective of energy and performance optimization.

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

1.1 Related Work

Both hardware and software approaches have been proposed to eliminate redundant computations in software programs [3, 4, 5, 6]. In this subsection, we discuss representative related work, and place our contributions in the proper context.

Value prediction [3] and *dynamic instruction reuse* [5] are hardware-assisted strategies for exploiting value locality in a program. Value prediction is a speculative technique that uses the result values from previous execution instances of an instruction to predict the value of the current execution instance without actual instruction execution. Dynamic instruction reuse, on the other hand, is a non-speculative technique that caches the inputs and results of previous execution instances of a computation. Due to hardware complexity limitations, these techniques exploit value locality only at the single instruction level, rather than at a larger granularity. The technique presented in [7] extended value reuse to basic blocks, where the effects of compiler optimizations have been evaluated. In [8], architecture and compiler support were integrated to realize the reuse of regions of computations. The instruction set architecture provides a simple interface for the compiler to communicate the scope of each reuse region and its register information to the hardware. However, the reuse regions generated in practice by this technique are still quite small, consisting of a few machine instructions [8].

The techniques mentioned above require architectural support to take advantage of value locality. The success of these techniques, together with the need to bridge the gap between the high computational burden and modest processing capabilities available on embedded systems, has fueled interest in exploring the use of value locality in embedded software. In [6], a software technique, called *memoization*, has been proposed to cache the function results to avoid unnecessary computations. It allows a computer program to run faster at the expense of increased memory usage. Once calculated, the result of a function is stored in a table called the memoization cache, which traditionally exists as a software data structure. Cache lookup then replaces later calls to the function. Impressive performance improvements have been reported by applying this technique to programs written in functional programming languages. Software caches have also been explored in the context of specific algorithms, *e.g.*, public-key encryption operations used in security protocols [9]. The results of highly computation-intensive operations, *e.g.*, modular exponentiation, and other modular arithmetic operations, are saved in a software cache, so that these operations can be skipped if the re-

sults can be found in the cache. However, these techniques are restricted to the reuse of either a single function or a single program statement. Therefore, they are incapable of identifying and exploiting general computation reuse regions that could consist of arbitrary sub-programs.

In this paper, we propose a profiling driven embedded software synthesis technique to eliminate redundant computations by reusing, or partially reusing, the computation results of a region of operations. We show that, even if computation results cannot be completely reused, partial computation reuse can lead to significant energy/performance improvements.

1.2 Paper Overview and Contributions

The contribution of our work is a systematic computation reuse methodology to eliminate redundant computations during embedded software synthesis, with an aim of improving the energy efficiency of the resulting implementations. Although not explicitly targeted, our methodology typically also results in *simultaneous* performance improvements.

Starting with an embedded software program to be optimized, and typical input traces that are used to profile the program and generate various statistics, the proposed methodology consists of the following steps: (i) value profiling of the input program to generate necessary statistics, (ii) identification of the regions of the software program that hold a high potential for computation reuse, (iii) identification of opportunities for partial computation reuse within the program regions identified in the previous step, (iii) exploration of various caching strategies or policies, and software cache parameters, which result in maximum energy savings, and (iv) transformation of the original program to include code that implements the software cache and performs cache lookups and updates.

We empirically demonstrate that judicious computation reuse leads to significant improvements in energy efficiency as well as performance, which are both processor-independent, and complementary to conventional compiler optimizations.

2 Motivation

In this section, we illustrate the basic ideas, and detail the tradeoffs and issues involved in our software synthesis technique, through illustrative examples. Section 2.1 shows the issues involved in performing computation reuse. Section 2.2 demonstrates that many partial results are reusable, even when complete result reuse is not possible.

2.1 Computation Reuse

In this section, we introduce the concept of region-level computation reuse with an illustrative example.

Example 1: Consider the example behavior shown in Figure 1(a), which is one of the MediaBench benchmarks [10] that converts uncompressed video frames into MPEG-2 video-coded bitstream sequences (MPEG-2 encoder), and decodes the MPEG-2 video frame (MPEG-2 decoder). The function shown in Figure 1(a), *Reference_IDCT*, which implements the double precision *inverse discrete cosine transformation (IDCT)*, is a core part of the MPEG-2 decoder. Array *c*[][] represents an 8×8 coefficient matrix, which is constant during the decoding process. The video frame (8×8 matrix) to be converted is stored in *block*[][]. Results

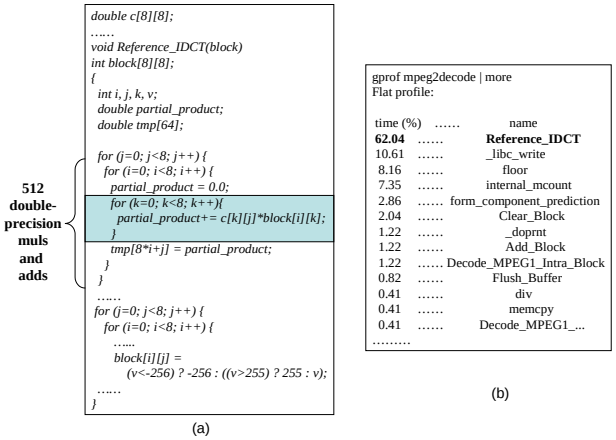

Figure 1: The `idct` example: (a) behavioral description, and (b) profi ling information

from *gprof* [1] shown in Figure 1(b) indicate that function *Reference_IDCT* accounts for a dominant portion of the execution time for the MPEG-2 decoder, *i.e.*, 62.04% of the total execution time. Therefore, the execution time for *Reference_IDCT* is a bottleneck for the overall program performance. A closer examination of the program indicates that the nested loop shown in Figure 1(a), which implements the double precision matrix multiplication of *c*[][] and *block*[][], consumes most of the execution time. There are 512 ($8 \times 8 \times 8$) double precision multiplications and additions to be executed each time the loop is executed. Therefore, the key to improving the performance is to reduce the amount of such expensive computations.

We profiled the benchmark with typical video frames provided along with the MediaBench benchmarks [10]. The profiling statistics indicate that the values read from the input video frame (*block*) are frequently the same or very similar to the ones from previous instances. More specifically, each row of the input *block*, *e.g.*, $block[i][0], block[i][1]$, ..., $block[i][7]$ (i^{th} row of the matrix), tends to repeat the values of the previous rows (either in the same matrix or previous matrices). Therefore, the inner loop, shown in the shaded region, which implements the multiplication of a constant column from *c*[][] and a row from *block*[][], often generates the same results. This observation motivates us to reuse the results generated by these operations. Based on the techniques described later in Section 3, this inner loop is chosen for computation reuse. The inputs of the chosen region are a row of the input matrix (*block*[][]) and a column of the constant coefficient matrix (*c*[][]), and the output is the *partial_product* variable.

Figure 2(a) shows the program optimized for computation reuse, and Figure 2(b) shows the flow of how we can avoid redundant operations by caching the computation results. The results of the shaded computations are stored in the cache, which is implemented as a software data structure. Before the operations in the shaded region are actually executed, the cache is searched to find whether the results are available. For a hit ($rt = 1$), we read the results from the cache into variable out and avoid the redundant computations. For a miss ($rt = 0$), we need to execute the original computations and update the cache accordingly. Profiling results indicate that the average cache hit rate for the selected

[1] gprof is a commonly-used UNIX profi ler for analyzing the distribution of execution time in a program.

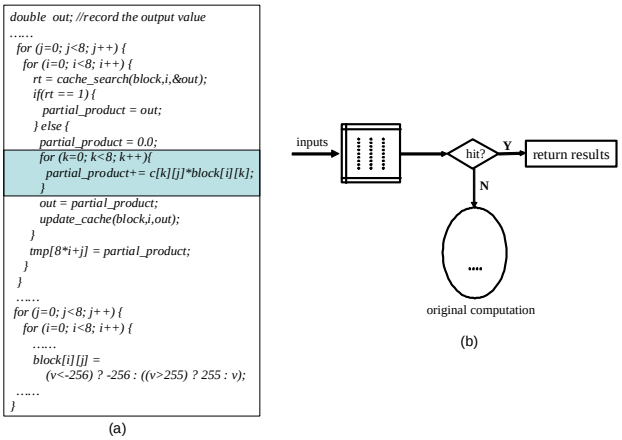

Figure 2: Computation reuse for `idct`: (a) optimized program, and (b) execution flow

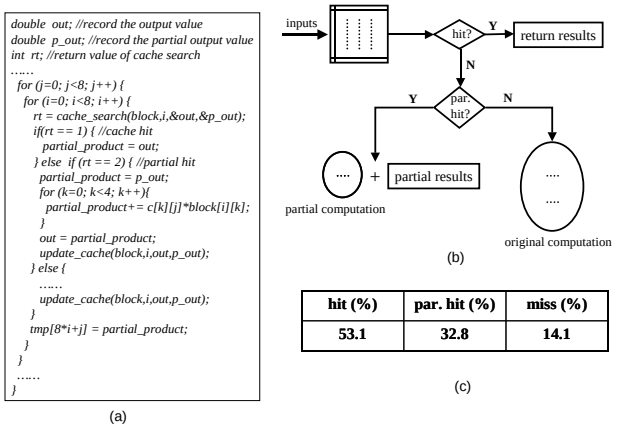

Figure 3: Partial result reuse: (a) optimized program, (b) execution flow, and (c) cache hit information

reuse region is 53.1%. Therefore, an average of 272 double precision multiplications and additions (out of a total number of 512) are skipped every time the outer loop is executed. We evaluated this benchmark running on the *Fujitsu SPARClite* based embedded system and the *iPAQ PDA*. The experimental results indicate energy reductions of 12.2% and 10.0% for the *Fujitsu SPARClite* and *iPAQ*, respectively. Simultaneously, performance improved by 12.0% and 18.4%, respectively. ∎

Based on the above example, the following aspects of our approach are worth noting. i) The reuse region chosen for optimization should be such that significant reduction in computation complexity occurs upon a cache hit. ii) We should perform computation reuse at a suitable granularity. Reuse at too small a granularity, *e.g.*, region with only one *add* operation, may yield very little or even negative improvement, due to the cache overheads. On the other extreme, the reuse at too large a granularity, *e.g.*, the whole function, may not occur frequently, again resulting in little savings. iii) The probability of reusing the previous results should be significant.

The above observations point to the need for a comprehensive methodology to efficiently reuse the computation results. We postpone the discussion of the methodology to Section 3, and instead shift our attention to partial result reuse.

2.2 Partial Result Reuse

In this section, we present the new concept of partial result reuse and show how it can further improve program efficiency with an illustrative example.

Example 2: Consider again the original program shown in Figure 1(a). For the target reuse region in the shaded area, the profiling statistics indicate that, even if the complete results cannot be reused, we can still benefit from partial results. This happens when only the last four values, *e.g.*, $block[i][4], ..., block[i][7]$, of the input row from *block*, are identical to the previous values stored in the cache. We call this a *partial cache hit*. Figure 3(c) shows that the entire row repeats the previous values in the cache with a probability of 53.1%, while for only the last four elements, the probability is 85.9%, *i.e.*, a partial cache hit occurs with a probability of $85.9\% - 53.1\% = 32.8\%$. This observation leads us to consider reusing the partial results made possible by the partial cache hit. In Figure 1(a), the last four iterations of the inner loop are redundant in the case of a partial cache hit, and therefore, these operations can be skipped. Figure 3(a) shows the transformed program which incorporates the reuse of partial results. Figure 3(b) presents the flow of the program enhanced with partial computation reuse. We integrate the implementation for partial result reuse with the implementation of complete computation reuse. In each cache entry, both the partial computation results and the complete results are stored. Therefore, we do not need an independent software cache just for the purpose of reusing partial results. Compared to the flow for computation reuse, there are more possibilities for the return result of a cache search: cache hit ($rt = 1$), cache miss ($rt = 0$) and partial cache hit ($rt = 2$). On a cache hit, *i.e.*, the complete result is reusable, we read the results from the cache and skip the redundant computations, as shown in Figure 2. However, if the complete result is not reusable, we still examine if there are any reusable partial results. If there are ($rt = 2$), we read the partial results from the cache into variable p_out and skip the computations made redundant by the partial cache hit. Otherwise ($rt = 0$), we execute the original computations and update the cache, as we do in Figure 2. Therefore, by employing partial result reuse, an additional 84 ($256 \times 32.8\%$) double precision multiplications and additions can be eliminated on an average. We evaluated the program with partial result reuse, running on the *SPARClite* and *iPAQ* systems. Experimental results indicate energy reductions of 15.6% and 15.3% for the *SPARClite* and *iPAQ*, respectively. Simultaneously, the performance is improved by 15.3% and 24.9%, respectively. ∎

Example 2 shows that there exist significant amounts of partial computation results in selected reuse regions. Considerable improvement can be achieved from such partial result reuse. However, it bears mentioning that a suitable implementation for partial result reuse is essential. In our technique, we integrate the implementation of complete and partial result reuse by storing the complete and partial results in the same cache. Therefore, we avoid implementing two independent caches for complete result reuse and partial result reuse, respectively, the overhead of which would have been significant.

3 Methodology and Algorithms

In this section, we present an overview of the methodology and algorithmic details of the proposed technique. Section 3.1 presents

the overall methodology and Section 3.2 explains the important steps in detail.

3.1 Overview

Figure 4 presents an overview of the proposed methodology. The inputs to the methodology are the input program and typical input traces.

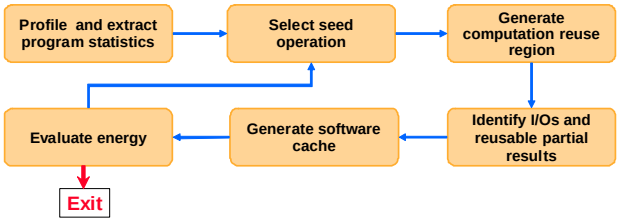

Figure 4: The overall methodology for computation reuse and partial result reuse

The first step is to profile the program with the input traces. In addition to extracting the control-flow statistics, we also collect the value profile [11] for each operation. Based on the statistics of the program, a starting operation, *seed*, is selected for generating the computation reuse region. To qualify as a *seed*, the operation must exhibit high potential for reuse. The details of the metric to select the *seed* operation are presented in Section 3.2.2. After selecting the *seed*, the reuse region is formed by repeatedly grouping neighboring operations, which meet certain constraints, to make the reuse of the region more beneficial. The target reuse region is then analyzed to find the inputs and outputs of the region, as well as any partial results, the reuse of which brings more benefit. Based on the above information, the cache, implemented as a software data structure, is generated to realize computation reuse. The details involved in cache implementation, *e.g.*, cache policy, cache size, *etc.*, are described in Section 3.2.3. The effect of reusing the selected region is then evaluated for energy and/or performance improvement. This is performed by adapting software power and energy models that have been proposed in the literature [12]. If we benefit from reuse, the above steps are repeated to generate more potential reuse regions. Otherwise, we exit the process and save the best solution.

3.2 Details

In this section, we give further details of the above-mentioned steps. Section 3.2.1 gives the profiling information for our technique. Section 3.2.2 discusses the formation of reuse regions. Section 3.2.3 describes the generation of the cache to capitalize on the reuse made possible by redundant computations.

3.2.1 Profiling

To realize computation reuse, it is necessary to capture run-time behavior of a program. In our work, we have used the *IMPACT* compiler [13] to extract necessary profiling results as well as useful dependencies embedded in the program.

The profiling traces are used to: i) evaluate the reuse potential of a single operation, ii) evaluate the reuse potential for a target reuse region, and iii) identify reusable partial results in a selected reuse region. In (i), we extract the output profile of the operation from the profiling trace, and calculate the important metrics for determining the reuse potential, *e.g.*, *reuse rate* and *average reuse distance*, whose details are presented in Section 3.2.2. In (ii), we evaluate the reuse potential for a reuse region in a way similar to (i), except that it is performed at a higher granularity. After the reuse region is identified, we still need to identify the partial results embedded in the target region, the reuse of which could be beneficial. Therefore, our technique is a profiling-driven region-level reuse strategy aimed at optimizing embedded software programs.

3.2.2 Reuse Region Formation

As shown in Figure 4, identification of the reuse region is one of the key steps in our technique. We employ a *bottom-up* method for this purpose. To form a candidate reuse region, we start with a single operation, and continue to incorporate neighboring operations until no further benefit can be achieved.

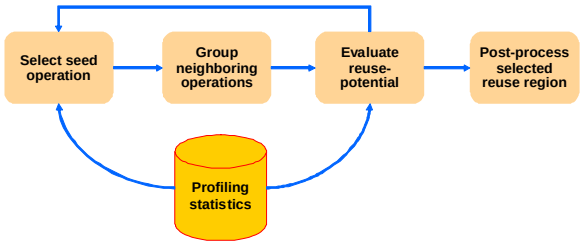

Figure 5: Identifi cation of the reuse region

Figure 5 presents an overview of our algorithm for forming a reuse region. From the profiling statistics, we first select a *seed* operation, the starting operation for forming a reuse region. To qualify as a seed, the operation must exhibit sufficient reuse potential. More specifically, its input values should repeat frequently across its execution instances. In addition, it is better to repeat input values of execution instances that are closer in time. We use two metrics, *reuse rate (RR)* and *reuse distance (RD)*, as defined below, to capture the above-mentioned factors.

Definition 1: RR(OP) *refers to the frequency with which the inputs of operation* OP *repeat the previous values:*

$$RR(OP) = \frac{M - J}{M} \quad (1)$$

where *J* represents the total number of distinct input instances and *M* represents the total number of input instances.

Definition 2: RD(OP) *refers to the number of instances between the current input instance and the nearest previous input instance that has the same value for operation* OP. *For the first input instance and input instances whose values never appear before,* RD(OP) *is equal to the total number of input instances, M. The average reuse distance,* ARD(OP), *is the average* RD(OP) *for the inputs of operation* OP *over all input instances:*

$$ARD(OP) = \frac{\Sigma_{i=1}^{M} RD_i(OP)}{M} \quad (2)$$

For example, if the current input instance is the fifth instance and the nearest previous instance with the same input value is the first instance, then $RD = 4$.

The combination of *ARD* and *RR*, which we term *Reuse_potential*, is defined in Equation (3).

$$Reuse\_potential(OP) = \frac{RR(OP)}{ARD(OP)} \quad (3)$$

$$= \frac{M-J}{\Sigma_{i=1}^{M} RD_i(OP)} \quad (4)$$

A high *Reuse_potential* indicates that the operation repeats its input values every few instances, and hence, is suitable for caching purpose. Therefore, *Reuse_potential* measures the desirability of an operation to be reused by our technique, and is used to select the seed operation.

After the seed operation is selected, the next step is to incorporate neighboring operations to form a reuse region. As discussed in Section 2, a target reuse region should contain significant amounts of computation, and the probability for reusing the results of the target region should be high. We define the *Reuse_potential* at the region level for a region σ in Equations (5) and (6), where *ARD* and *RR* are now computed at the region level. w_i is the weight of the i^{th} operation, and Num is the total number of operations in σ. The sum of the weights in the target region reflects the computational complexity of the region, and is referred to as total weight (*TW*). $f(\sigma)$ is a function that measures the impact of the number of inputs and outputs of the reuse region on the overhead of cache lookup and update. Generally speaking, a larger number of inputs and outputs leads to more overhead during cache lookup and update. The exact form of this function is derived from experimentally-obtained statistics, gathered from the cache implementation. A region with a higher *Reuse_potential* will have a better chance to be selected as the target. Therefore, an operation is grouped into a reuse region if its *Reuse_potential* can be improved. We stop the grouping process when *Reuse_potential* is maximized and adding further operations only results in a reduction in its value.

$$Reuse\_potential(\sigma) = \frac{RR(\sigma) \times TW \times f(\sigma)}{ARD(\sigma)} \quad (5)$$

$$= \frac{M-J}{\Sigma_{i=1}^{M} RD_i(\sigma)} \times \Sigma_{i=1}^{Num} w_i \times f(\sigma) \quad (6)$$

The reuse region selected above is post-processed if necessary. In this step, we examine whether or not there are program side effects. Side effects, *e.g.*, change of global variables or memory content, are prohibited in a reuse region to ensure correct functionality.

3.2.3 Cache Generation

It is not surprising that different cache sizes and replacement policies have different impact on energy and performance. Therefore, a suitable cache size and replacement policy need to be determined in our technique. There exist two types of techniques for determining the cache parameters: static analysis and simulation-based [14]. The static analysis techniques have been shown to be unable to adapt to changes in inputs [14]. Therefore, in our methodology, we employ a simulation-based technique.

Figure 6 shows our strategy for selecting the cache size and replacement policy. For a target reuse region, we simulate the candidate cache sizes and replacement policies with a virtual cache, which only keeps the object header information, rather than the actual data. The candidate cache sizes are selected based upon

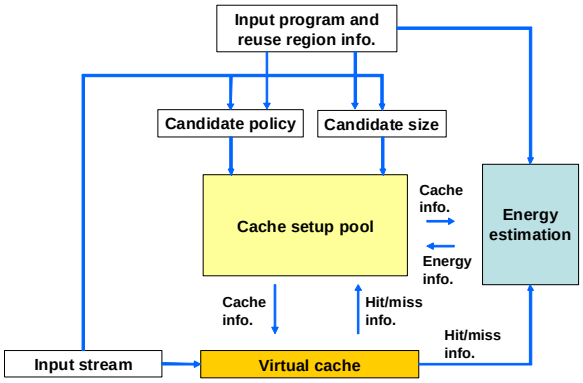

Figure 6: Cache simulation system

profiling statistics, while the candidate replacement polices include most of the popular policies, *e.g.*, first-in first-out (FIFO), least frequently used (LFU), least recently used (LRU), *etc.* The hit and miss events occurring as a result of the input stream and cache setup are fed back to the cache configuration system. From the information available in the input program and reuse region, as well as the hit/miss rate of the candidate cache setup, we are able to know the number of times each operation has been executed, as well as the number of cache updates. Therefore, an estimate of the energy can be given by employing the energy model proposed in [12].

4 Experimental Evaluation

We applied our technique to several embedded software benchmarks. Typical input traces were available for all the programs. The programs were optimized by applying the procedure described in Section 3. The original and optimized programs were compared with respect to the following metrics: *performance*, *energy* and *energy-delay product*.

4.1 Experimental Setup

We evaluated the energy consumption and performance of the programs for two systems: an embedded system based on the *Fujitsu SPARClite* processor, and the *Compaq iPAQ Pocket PC* (64MB memory, 206MHz Intel StrongARM CPU). For *SPARClite* based system, instruction-level energy estimation was performed to evaluate energy and execution time savings. The details of the energy estimation tool can be found in [15]. Therefore, we focus here on the setup used to measure the energy consumption of the *iPAQ*.

As shown in Figure 7, the *iPAQ* communicates with a PC using its *general-purpose input-output (GPIO)* interface. The PC reads the communication data, which includes the current consumption value ($I(t)$), the *start* and *stop* signals, *etc.*, from the *data acquisition (DAQ)* board [16]. The *Labview* software [17], which is installed on the PC, is used to analyze the acquired data and perform necessary computations to get the energy and performance results. The power consumption of the *iPAQ* is calculated as the product of V_{dd} and $I(t)$, where V_{dd} is the constant system voltage, 3.3V, and $I(t)$ is the current flowing from the *iPAQ*'s power supply. To get the energy results, we need to identify the time interval when the benchmark program is actually executing. Therefore, at the start and end of execution, a *start* and *stop* signal are sent from the *iPAQ* to the PC, from which we can determine the

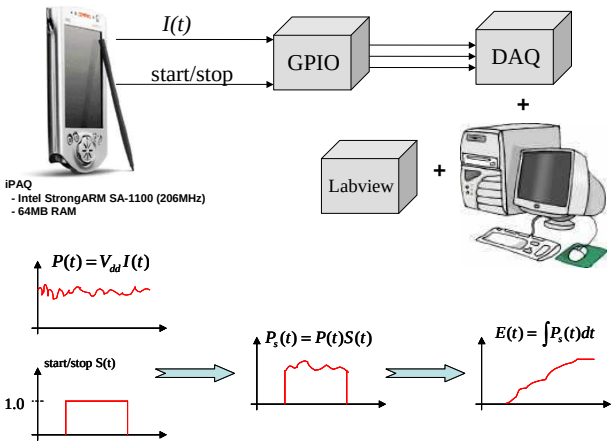

Figure 7: Setup for measuring energy of the *iPAQ*

time interval of execution ($S(t)$). The power computed above is integrated over this time interval to get the energy consumption.

4.2 Experimental Results

The normalized energy and performance results obtained are shown in Figures 8 and 9, respectively. Of our benchmarks, `idct`, which implements the inverse discrete cosine transformation, was discussed in Section 2. `dct` represents the discrete cosine transformation. `qt1` and `qt2` represent functions that perform quantization in intra-frame and inter-frame coding in MPEG-2 encoder, respectively. `adpcm` represents a program extracted from the adaptive differential pulse code modulation application. `cubic` is a program for solving a cubic equation.

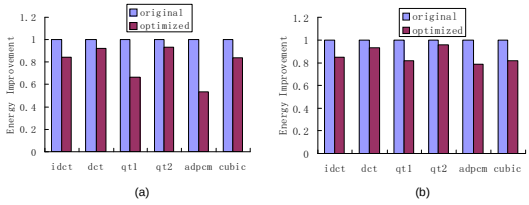

Figure 8: Energy results for (a) *Fujitsu SPARClite*, and (b) *iPAQ*

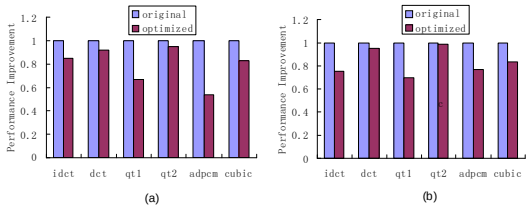

Figure 9: Performance results for (a) *Fujitsu SPARClite*, and (b) *iPAQ*

By employing our technique, the average energy reductions for *SPARClite* based system and *iPAQ* are 21.2% and 13.9%, respectively. Note that our techniques result in *simultaneous* energy savings and performance improvements. Hence, we also present performance results. The average performance improvements for *SPARClite* based system and *iPAQ* are 20.7% and 16.8%, respectively. The energy-delay product for the programs (product of the *energy* and *execution time*) have also been computed. The product is reduced by an average of 35.4% for the *SPARClite* based system, and 26.9% for the *iPAQ*.

5 Conclusions

In this paper, we presented a design methodology for optimizing performance and energy consumption through complete or partial reuse of computation results. We presented algorithms for performing the various steps in computation reuse, including selection of reuse regions and generation of the software cache to store the computation results. Experimental results on several benchmarks demonstrated that significant amounts of redundant computations exist in embedded software programs, whose elimination can lead to *simultaneous* improvements in performance and energy.

References

[1] G. K. Zipf, "Relative frequency as a determinant of phonetic change," *Reprinted from the Harvard Studies in Classical Philiology*, vol. XL, 1929.

[2] L. Breslau, P. Chao, L. Fan, G. Philips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. IEEE INFOCOM*, pp. 126–134, Mar. 1999.

[3] T. Nakra, R. Gupta, and M. Soffa, "Value prediction in VLIW machines," in *Proc. Int. Symp. Computer Architecture*, pp. 258–269, May 1999.

[4] M. H. Lipasti and J. P. Shen, "Value locality and load value prediction," in *Proc. Int. Conf. Architecture Support for Programming Languages & Operating Systems*, pp. 138–147, Sept. 1996.

[5] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proc. Int. Symp. Computer Architecture*, pp. 194–205, June 1998.

[6] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," Tech. Rep. 92-1, Sun Microsystems Laboratories, Sept. 1992.

[7] J. Huang and D. J. Lilja, "Extending value reuse to basic blocks with compiler support," *IEEE Trans. Computers*, vol. 49, pp. 331–347, Apr. 2000.

[8] D. Connors and W. M. Hwu, "Compiler directed dynamic computation reuse: Rationale and initial results," in *Proc. Int. Symp. Microarchitecture*, pp. 158–169, Nov. 1999.

[9] N. R. Potlapally, S. Ravi, A. Raghunathan, and G. Lakshminarayana, "Algorithm exploration for effi cient public-key security processing on wireless handsets," in *Proc. Design Automation & Test Europe Conf.*, pp. 42–46, Mar. 2002.

[10] Media applications and data (http://www.cares.icsl.ucla.edu/MediaBench).

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

[12] W. Wang, A. Raghunathan, G. Lakshminarayana, and N. K. Jha, "Input space adaptive embedded software synthesis," in *Proc. Int. Conf. VLSI Design*, pp. 711–718, Jan. 2002.

[13] IMPACT Advanced Compiler Technology (http://www.crhc.uiuc.edu/Impact).

[14] I. Ari, A. Amer, E. Miller, S. Brandt, and D. Long, "Who is more adaptive? ACME: Adaptive caching using multiple experts," in *Proc. Wkshp. Distributed Data & Structures*, Mar. 2002.

[15] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Analysis of power dissipation in real-time operating systems," *IEEE Trans. Computer-Aided Design*, vol. 22, pp. 615–627, May 2003.

[16] *DAQ Hardware Overview Guide*. National Instruments, Inc., Oct. 2000.

[17] *LabVIEW User Manual*. National Instruments, Inc., Nov. 2001.