Challenges in Exploitation of Loop Parallelism in Embedded Applications

Arun Kejariwal[†] Alexander V. Veidenbaum[†] Alexandru Nicolau[†]
[†]Center for Embedded Computer Systems
University of California at Irvine
Irvine, CA, USA

Milind Girkar[‡] Xinmin Tian[‡] Hideki Saito[‡]
[‡]Intel Corporation
3600, Juliette Lane
Santa Clara, CA, USA

ABSTRACT

Embedded processors have been increasingly exploiting hardware parallelism. Vector units, multiple processors or cores, hyper-threading, special-purpose accelerators such as DSPs or cryptographic engines, or a combination of the above have appeared in a number of processors. They serve to address the increasing performance requirements of modern embedded applications. How this hardware parallelism can be exploited by applications is directly related to the amount of parallelism inherent in a target application. In this paper we evaluate the performance potential of different types of parallelism, viz., true thread-level parallelism, speculative thread-level parallelism and vector parallelism, when executing loops. Applications from the industry-standard EEMBC 1.1, EEMBC 2.0 and the MiBench embedded benchmark suites are analyzed using the Intel C compiler. The results show what can be achieved today, provide upper bounds on the performance potential of different types of thread parallelism, and point out a number of issues that need to be addressed to improve performance. The latter include parallelization of libraries such as libc and design of parallel algorithms to allow maximal exploitation of parallelism. The results also point to the need for developing new benchmark suites more suitable to parallel compilation and execution.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Programming Techniques—*languages; methodolgies*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance, Measurement

Keywords

Multi-cores, Programming models, Libraries, Multithreading, Vectorization, Thread-level speculation, Parallel loops

This work was supported in part by the National Science Foundation under grants NSF CCF-0311738 and CNS-0220069.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'06, October 22–25,2006,Seoul,Korea.
Copyright 2006 ACM 1-59593-370-0/06/0010 ...$5.00.

1. INTRODUCTION

In recent years many hardware techniques have been incorporated into embedded processors for exploiting parallelism. These include vector units, multiple processors or cores, hyper-threading, special-purpose accelerators et cetera. There are several reasons for this trend. One such reason is the increasing transistor budgets which has enabled the realization of multi-core processors; another reason is the increasing emphasis on low power design. The trend is further stimulated by the increased performance requirements in many application areas, e.g., networking, xDSL, security, wireless and game applications et cetera.

This has led to the development of heterogeneous and application-specific MpSoCs [1, 2, 3, 4, 5] and embedded processors with vector capabilities [6, 7] or multithreading capabilities [8]. Such systems may also be augmented with co-processors which serve as accelerators for application-specific functions. For example, Intel's IXP2850 network processor [8] has integrated with cryptography engines to facilitate fast encryption/decryption; TI OMAP chips have an integrated DSP co-processor [9].

These trends are in fact similar to those in the high-performance processor design, e.g., Intel's dual-core Yonah processor [10] or the Intel's dual-core and hyper-threaded Xeon® processor [11] and the IBM/Sony/Toshiba Cell processor [12] with its eight specialized SIMD units for data-intensive processing. The trend towards integrating more and more cores on an MpSoC is ramping up [13]. Some trends in this domain have not yet migrated to the embedded processor domain, e.g., speculative multithreading [14], but it may be only a matter of time.

The use of aforementioned hardware techniques helps to achieve better performance than a standard uniprocessor by exploiting hardware parallelism. However the above is valid only for applications that are written or compiled for exploiting the available hardware parallelism. The actual improvement is limited by a "mismatch" between the type of parallelism — Instruction-level parallelism (ILP), SIMD (single instruction multiple data) parallelism, MIMD (multiple instruction multiple data) [15] or true thread-level parallelism (TLP), speculative thread-level parallelism (sTLP) — inherent in a given application and the type of parallelism supported by the underlying hardware. Thus the improvement is limited by how well the compiler can generate code for "matching" the hardware parallelism and the application parallelism. In case of applications which contain multiple types of parallelism, it is even more challenging for the compiler to efficiently exploit the available parallelism.

Loop-level parallelization has been one of the most widely used techniques for program parallelization. This can be attributed to the fact that in most applications loops account for a large percentage of the total execution time. Given a multi-core, the iterations of a `DOALL` loop[1] are executed in parallel by mapping them onto different threads. This corresponds to an instance of exploitation of TLP. On the other hand, loops with dependences between its iterations (referred to as `Non-DOALL` loops in the rest of the paper) can be parallelized either speculatively or with explicit synchronization. The former corresponds to an instance of exploitation of sTLP.

In this paper we evaluate the performance potential of different types of parallelism in embedded applications. For this, we perform the performance evaluation at the loop level. Due to space limitations, we only present analysis for the innermost loops. The loop coverage, defined as the percentage of the total execution time spent in the loops, is obtained by first instrumenting the code of each application during compilation with hardware performance counters and then executing it on an 3.6 GHz Intel® Xeon® Processor. The reason for using the Pentium processor instead of an embedded processor is that it integrates vector and multithreading support and the availability of an auto-parallelizing compiler.

The main contributions of the paper are as follows:

- First, a loop-level characterization of the industry-standard EEMBC 1.1, EEMBC 2.0 and the MiBench embedded benchmark suites is presented.
- Second, the performance potential of true TLP is estimated. In other words, an optimistic upper bound on the speedup achievable (at the loop-level) via auto-parallelization [17] is determined. Given an application, this provides an estimate of the number of cores required for maximal exploitation of TLP.
- Third, the performance potential of sTLP is estimated. In other words, an optimistic upper bound on the speedup achievable via thread-level speculation (TLS) is determined.
- Fourth, the impact of vectorization on performance is evaluated. To our surprise, we find that the amount of SIMD parallelism in different classes of application space, except multimedia applications, is rather limited. This is explained, in part, by the compiler's choice of exploiting TLP or MIMD parallelism.
- Fifth, we identify limitations of parallelization of the innermost loops and suggest ways to alleviate them with user and compiler assistance.

This type of analysis exposes the parallelism inherent in a given application. The analysis can be used in a variety of ways. A hardware designer can use it to (a) make design decisions such as deciding between multiple cores, multithreading support, or having vector units, based on the performance, power and cost trade-offs; (b) design application-specific processors. Application developers can use it to modify programs to better exploit a given type or types of hardware parallelism or to assist the compiler by giving "hints" to the compiler in the form of directives/pragmas which guide the compiler to generate better parallel code. The compiler writers can use it (i) to develop better code generation strategies; (ii) for profitability analysis, and (iii) to develop ways to exploit multiple types of hardware parallelism. For example, in `susan` (an application in MiBench), one observes that parallel (non-vector) loops account for 68.4% of the total execution time. In contrast, vector loops account account for less than 1% of execution time. This is in part due to the code generation strategy that puts a premium on MIMD parallelism and did not have any vector parallelism "left over". Given efficient support for MIMD parallelism (or TLP) in loops it may not be useful to provide additional hardware support for vector execution in this case. Instead, it is better to increase the number of cores or provide multithreading support for exploiting TLP.

[1]A `DOALL` loop is a loop in which there does not exist a control or data dependence between any two iterations of the loop [16].

The rest of the paper is organized as follows: Loop-level evaluation of the performance potential of the different types of parallelism is presented in Section 2. Specifically, loop-level characterization of EEMBC 1.1, 2.0 and MiBench suites is presented in subsection 2.1, the evaluation of the performance potential of TLP and sTLP is presented in subsection 2.2 and the evaluation of the impact of vectorization on performance is presented in subsection 2.3. Finally, we conclude in Section 3.

2. LOOP-LEVEL PARALLELISM ANALYSIS

In this section, we first present a loop-level characterization (as loop parallelization has been one of the most widely used techniques for program parallelization) of the EEMBC 1.1, EEMBC 2.0 (Networking) and Mibench (described below). To our knowledge, this is the first characterization of this kind for these suites. Note that no source code changes were made in either of the two benchmark suites during the performance evaluation. Next, we discuss the loop-level speedup achievable by exploiting loop-level TLP and sTLP. Finally, we estimate the performance potential of vectorization for embedded applications.

Let us start with a brief overview of the applications in the industry-standard embedded benchmark suites — EEMBC 1.1 and EEMBC 2.0 [18] and the academic embedded benchmark suite MiBench 1.0 [19]. Both EEMBC 1.1 and MiBench are divided into multiple classes which are representative of different embedded application domains such as automotive, consumer, networking, office (see Tables 1 and 3). The EEMBC 2.0 benchmark suite is currently under development; as of now, it has applications from the networking domain only (see Table 2). All benchmarks are written in the C language. Interestingly, benchmarks range from a modest 100 lines of code to $\approx 208K$ lines of code. As will be shown in the next section, memory and I/O operations account for a large percentage of the total execution time in many benchmarks. From Tables 1, 2 and 3 we note that the suites differ in the selection of the benchmarks. In view of the above, we carried performance analysis of all the three suites so as to cover a wider spectrum of applications.

The results presented in the rest of this section were obtained by running the benchmarks with the reference data sets (large data sets in case of MiBench) on the Intel® Xeon® Processor. The configuration details of the system are given in Table 4. The benchmarks were compiled using the Intel® C++/Fortran optimizing compiler. The compiler performs

Class	Benchmark	Lines of Code	Language	Description
8_16-bit	bitmnp816	2272	C	Bit Manipulation
	canrdr816	3811	C	Response to Remote Request (CAN)
	memacc816	1715	C	Memory Access
	pntrch816	1574	C	Pointer Chasing
	puwmod816	3967	C	Pulse Width Modulation Initialization
	rspeed816	2326	C	Road Speed Calculation
	ttsprk816	25310	C	Tooth-to-Spark
automotive	a2time01	3455	C	Angle-to-Time Conversion
	aifftr01	10371	C	Fast Fourier Transform
	aifirf01	4385	C	Finite Impulse Response
	aiifft01	10198	C	Inverse Fast Fourier Transform
	basefp01	5907	C	Basic Integer and Floating-Point
	bitmnp01	3928	C	Bit manipulation
	cacheb01	2333	C	Cache Buster
	canrdr01	5782	C	CAN Reader algorithm
	idctrn01	18637	C	Inverse Discrete Cosine Transform
	iirflt01	4988	C	Low-Pass Filter (IIR) and DSP functions
	matrix01	3029	C	Matrix Math
	pntrch01	1928	C	Pointer Chasing
	puwmod01	7323	C	Pulse-Width Modulation
	rspeed01	2713	C	Road Speed Calculation
	tblook01	1792	C	Table lookup and interpolation
	ttsprk01	28348	C	Tooth to Spark
consumer	cjpeg	50292	C	Jpeg Compression
	djpeg	17766	C	Jpeg Decompression
	rgbhpg01	26191	C	Image filter
networking	ospf	2015	C	A benchmark based on Dijkstra's shortest-path algorithm
	pktflow	2264	C	Packet Flow: Receives and processes incoming IP packets according to a subset of RFC1812, "Requirements for Routers"
	routelookup	1523	C	RouteLookup Benchmark
office	bezier01	3304	C	Bezier Interpolation
	dither01	27863	C	Floy-Steinberg Error Diffusion Dithering Algorithm
telecom	autocor00	997	C	Fixed Point AutoCorrelation
	conven00	1018	C	Convolutional Encoder
	diffmeasure	650	C	Measurement of Signal to Noise Difference in DB
	fbital00	1155	C	Fixed Point Bit Allocation
	fft00	1626	C	Fixed Point Complex FFT/IFFT
	viterb00	1286	C	Viterbi Decoder

Table 1: Description of benchmarks in EEMBC1.1

Class	Benchmark	Lines of Code	Language	Description
networking	ip_pktcheck	2264	C	Packet Flow: Receives and processes incoming IP packets according to a subset of RFC1812, "Requirements for Routers"
	ip_reassembly	2593	C	Packet reassembly
	nat	6955	C	Network Address Translation
	ospf	2049	C	A benchmark based on Dijkstra's shortest-path algorithm
	qos	6643	C	Quality of Service
	routelookup	1571	C	RoutLookup Benchmark
	tcp	3875	C	Transmission Control Protocol

Table 2: Description of benchmarks in EEMBC2.0

a large set of optimizations via procedural inlining, advanced inter-procedural analysis for maximal parallelization. Further, in order to alleviate the limitations of the compiler, we carried manual analysis (at the source level) of some of the loops which the compiler could not parallelize. Only (about) 10% of the total number of loops were parallelized manually. Such loops can be easily parallelized via, for example, OpenMP [20] pragmas.

2.1 Loop-level Characterization

In this subsection, we present the innermost loop coverage, defined as the percentage of the total execution time spent

Class	Benchmark	Lines of Code	Language	Description
automotive	basicmath	579	C	Basic math routines such as cube, sqrt
	bitcount	917	C	Bit counting functions
	qsort	100	C	Qsort Algorithm
	susan	2122	C	Smallest Univalue Segment Assimilating Nucleus (Image processing)
consumer	jpeg	33717	C	Jpeg Compression
	lame	18612	C	LAME Ain't an MP3 Encoder
	typeset	47185	C	Document Formatting System
networking	dijkstra	351	C	Dijkstra's algorithm
	patricia	599	C	Trie implementation
office	ghostscript	197528	C	Ghostscript 5.0
	ispell	12527	C	Spell Check
	rsynth	7055	C	Speech system
	sphinx	208810	C	Large vocabulary, speaker-independent continuous speech recognition engine
	stringsearch	3216	C	String search
security	blowfish	2302	C	A keyed, symmetric block cipher
	pgp	34858	C	Pretty Good Privacy version 2.6.3i
	rijndael	1788	C	Advanced Encryption Standard
	sha	269	C	Secure Hash Algorithm
telecom	adpcm	741	C	Adaptive Differential Pulse Code Modulation
	CRC32	281	C	Cyclic Redundancy Check
	FFT	469	C	Fast Fourier Transform

Table 3: Description of benchmarks in MiBench

Processor	Intel®Xeon™Processor 3.6 GHz
Memory	2 GB (PC2700 a.k.a. DDR333 DIMMs)
L2 Cache	1 MB
Compiler Flags	-O3 -ansi_alias -xN parallel
OS	Linux 2.4.21-12.EL #1 SMP

Table 4: Experimental Setup

in such loops, for applications in EEMBC 1.1, EEMBC 2.0 and MiBench. Note that the coverage numbers presented in this subsection correspond to only those loops that are present in the (optimized) application code; loops in the library functions are excluded. Also, the coverages correspond to single thread execution of the parallelized code. Of course, the coverages shown later in this subsection are subject to the particular algorithm selected and its implementation and the compiler used.

To obtain this, the code generator of the Intel® compiler was modified for automatic insertion of hardware performance counters. The point of insertion of these counters (amongst the different phases of the compilation process) has a direct effect on the coverage analysis. This is due to the fact that insertion of these counters early in the compilation process can potentially disable some of the optimizations. Therefore, it is critical to make sure that these counters are inserted only during the code generation phase. In our experiments, we account for the overhead incurred due to the insertion of these counters. A detailed discussion of our instrumentation support is beyond the scope of the paper.

The total loop coverage is computed as follows: first, *tick%*, defined as the total time spent in a given loop, is determined for each loop. Next, the *self%*, defined as the execution time spent in a loop excluding the time spent in any function or any other loop that may be embedded in it, is determined for each loop. Finally, the *self%* of all the loops is summed to obtain the total loop coverage for a given application.

The number of innermost loops executed and their coverage for select applications in EEMBC 1.1, 2.0 and MiBench are given in Tables 6, 5 and 7 respectively. In case of EEMBC 1.1 and EEMBC 2.0, results are shown only for those applications which were executed using the testing framework that comes with the corresponding suite.

From the table we observe that in some benchmarks the innermost loops have a low coverage. The reasons for this are discussed below.

- In many benchmarks the low loop coverage can be attributed to the large amount of time spent in library calls. For example, the I/O function `al_write_con` in the `ttsprk816` benchmark has a coverage of 17.25% (the benchmark `ttsprk816` has very low (< 1%) innermost loop coverage; due to this we do not list it in Table 6). However, there do not exist any loops in the function (see below). Most of the coverage of the function is due to the `fwrite` library call.

  Similarly, the library calls `vsprintf` and `strlen` ac-

Benchmark	ip_pktcheck	qos	tcp
# of loops	17	27	42
% Execution Time	43.9	54.1	23.4

Table 5: Total number of innermost loops executed (after optimization) and their coverage, for EEMBC 2.0

Benchmark	a2time01	canrdr01	cjpeg	djpeg	rgbhpg01	pktflow	bezier01	rotate01	autocor00	conven00	fbital00	fft00	viterb00
# of loops	10	13	107	110	30	19	12	11	17	15	6	22	16
% Execution Time	66.2	1.4	57.3	52.1	87	43.8	64.2	93.1	82	44.9	64.3	59	66.1

Table 6: Total number of innermost loops executed (after optimization) and their coverage, for EEMBC 1.1

Benchmark	basicmath	bitcount	qsort	susan	jpeg	lame	typeset	dijkstra	particia	ghostscript
# of loops	14	3	2	10	85	252	821	6	6	356
% Execution Time	95.5	38.7	52.8	80.6	25.4	49.1	15	70.1	95.1	75.1

Benchmark	ispell	rsynth	stringsearch	blowfish	pgp	rijandel	sha	adpcm	CRC32	FFT
# of loops	78	46	5	13	10	6	11	2	1	13
% Execution Time	16.7	60.4	81.1	38.1	92	7.8	66.9	76.3	99.5	94.1

Table 7: Total number of innermost loops executed (after optimization) and their coverage, for MiBench

```
int al_write_con (const char* tx_buf, size_t byte_count) {
  /* This logic must be preserved */
  if (byte_count ==  0)
    return Success;
  fwrite( tx_buf, sizeof(char), byte_count, stdout );
  return Success;
}
```

count for the entire coverage (= 35.65%) of the function `i_printf` of the `ttsprk816` benchmark. Further, the library calls `printf, strcmp` and `strcat` account for almost the entire coverage of the function `th_main`. In a similar vein, in many benchmarks, I/O accounts for a large part of the total execution time.

The above observation provides valuable guidance for design of high performance embedded systems and optimization of embedded applications: for applications such as `ttsprk816`, it is critical to design better I/O mechanisms and address optimization of library routines rather than optimizing the application itself, in order to achieve high performance.

- In benchmarks such as `canrdr01` the low coverage of innermost loops can be in part attributed to their very small (in the number of instructions) loop bodies. Also, there exist quite a few functions with large coverage but with no loops. For instance, the function `WriteOut` (shown below, taken from bmark.c:1182) accounts for 15.79% of the total execution time but has no loops.

```
n_void WriteOut( varsize value ) {
if (( RAMfilePtr+RAMfile_increment) > RAMfileEOF )
  RAMfilePtr = RAMfile;
  *(varsize *)RAMfilePtr = value;
  RAMfilePtr += RAMfile_increment;
} /* End of function 'WriteOut' */
```

Likewise, the function `GetInputValues` (algotst.c:3222) has a coverage of 19.22% and does not contain any loops.

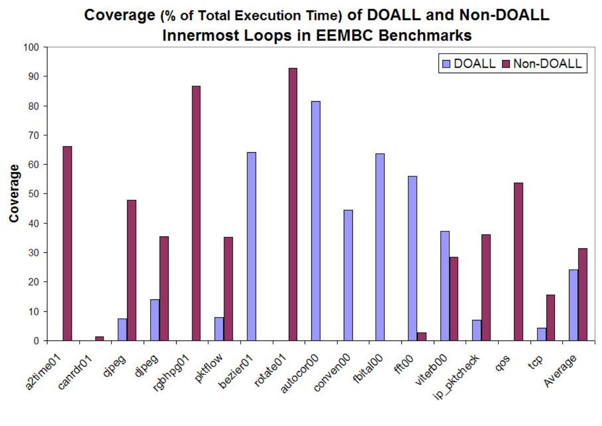
- In benchmarks such as `canrdr01` the outermost loops have a large coverage (*self%*). For instance, the function `t_run_test` (bmark.c:186) has a large coverage of 62.95%. The outermost loop at line 342 in this function accounts for most of the function's coverage. The aforementioned functions are called in this loop. This illustrates the need for exploitation of parallelism at higher levels such as at the outermost loop level, at function level. However, this may be difficult to exploit due to the I/O involved. Likewise, exploitation of hierarchical parallelism as in parallel *multi-way* loops [21] can potentially yield better performance. A discussion of techniques exploiting the above is beyond the scope of the paper.

- In applications such as `bitcount`, the low loop coverage is due to the fact that a large amount of time is spent in recursive execution. For instance, the function `ntbl_bitcnt` (bitcount/bitcnt_4.c:38) is implemented in a recursive fashion and has a large coverage. In such cases, conversion of recursion to loops may help program parallelization [22, 23]. Alternatively, parallel recursive algorithms must be used in such cases.

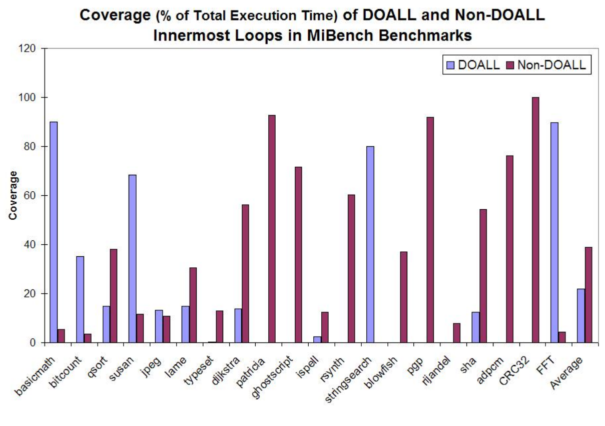
- Another reason for low loop coverage is that in many applications a large amount of time is spent in memory allocation. For instance, the function `nbuf_alloc` in the `tcp` benchmark accounts for 14.48% of the total execution time. `nbuf_alloc` allocates memory using the `memset` library call. Similarly, the function `tcp_memcpy` (which internally calls the `memcpy` function) has a coverage of 6.02%. To better understand this, let us analyze the code of the `memset` library code (taken from the GNU C library, version 2.4 [24], see below) for example.

In the code snippet, the while loop at line 23 accounts for most of the time of the entire function. Parallelization or SIMDization of this while loop can potentially lead to better performance.[2] On analysis, we see that

[2]The profitability of parallelization/SIMDization depends on various factors such as the number of iterations of the loop, SIMD support in the processor. This gives rise to a need for multi-versioning [25] of the while loop.



(a)



(b)

Figure 1: `DOALL`, `Non-DOALL` breakdown of the coverage of the innermost loops in EEMBC and MiBench

the while loop can be converted into a for loop and can then be parallelized using OpenMP pragmas in conjunction with IVE (applied on `dstp`).

```
void * memset (dstpp, c, len)
     void *dstpp; int c; size_t len;
{
  long int dstp = (long int) dstpp;
  if (len >= 8) {
      size_t xlen;
      op_t cccc;
      cccc = (unsigned char) c;
      cccc |= cccc << 8;
      cccc |= cccc << 16;
      if (OPSIZ > 4)
        /* Do the shift in two steps to avoid warning if long has 32 bits. */
        cccc |= (cccc << 16) << 16;
      /* There are at least some bytes to set.  No need to test for
         LEN == 0 in this alignment loop. */
      while (dstp % OPSIZ != 0) {
        ((byte *) dstp)[0] = c;
        dstp += 1;
        len -= 1;
      }
      /* Write 8 `op_t' per iteration until less than 8 `op_t' remain.  */
      xlen = len / (OPSIZ * 8);
23:   while (xlen > 0) {
        ((op_t *) dstp)[0] = cccc;
        ((op_t *) dstp)[1] = cccc;
        ((op_t *) dstp)[2] = cccc;
        ((op_t *) dstp)[3] = cccc;
        ((op_t *) dstp)[4] = cccc;
        ((op_t *) dstp)[5] = cccc;
        ((op_t *) dstp)[6] = cccc;
        ((op_t *) dstp)[7] = cccc;
        dstp += 8 * OPSIZ;
        xlen -= 1;
      }
      len %= OPSIZ * 8;
      /* Write 1 `op_t' per iteration until less than OPSIZ bytes remain.  */
      xlen = len / OPSIZ;
      while (xlen > 0) {
        ((op_t *) dstp)[0] = cccc;
        dstp += OPSIZ;
        xlen -= 1;
      }
      len %= OPSIZ;
  }
  /* Write the last few bytes.  */
  while (len > 0) {
      ((byte *) dstp)[0] = c;
      dstp += 1;
      len -= 1;
  }
  return dstpp;
}
```

The above exemplifies the importance of parallelization of libraries and is part of our current research.

2.2 TLP vs. sTLP

Recall that parallel execution of `DOALL` loops corresponds to exploitation of TLP, whereas speculative parallel execution of `Non-DOALL` loops corresponds to an instance of sTLP. Figure 1 shows the `DOALL`/`Non-DOALL` breakdown of the coverage of innermost loops for EEMBC 1.1, EEMBC 2.0 and MiBench respectively. The coverage of all `DOALL` loops corresponds to an upper bound on the speedup achievable via TLP. In other words, assuming an oracle TLP mechanism whereby the execution time of a candidate loop can be reduced to zero, the speedup achievable via TLP is equal to the total coverage of the `DOALL` loops. However, in practice the speedup achievable via TLP is limited by many factors such as the threading overhead. Thus, the coverages shown in Figure 1 are very optimistic upper bounds.

From Figure 1(a) we see that in 6 out of 16 applications `DOALL` loops account for most of the total loop coverage. On an average, TLP has a performance potential of 24%, which is rather small. On the other hand, from Figure 1(b) we note that in only 5 out of 20 applications `DOALL` loops account for most of the total loop coverage. On an average, TLP has a performance potential of 22%. In view of the increasing emphasis on putting more cores on a chip, the above gives rise to a need to revisit design of parallel algorithms and parallel programming models.

From Figure 1 we see that `Non-DOALL` innermost loops have a large coverage — 31% in EEMBC 1.1, 2.0 and 39% in MiBench, on an average. This corresponds to an upper bound on the speedup achievable via sTLP. However, this is a loose upper bound as threaded execution of some loops can potentially result in performance degradation because of the threading and misspeculation overhead. Most of these `Non-DOALL` loops are `DO` loops with inter-iteration dependences or are true `WHILE` loops. Various schemes, viz., data dependence speculation (DDS), control speculation (CS) and data value speculation (DVS) have been proposed for extracting parallelism from such loops.[3] For example, the loop at line bmark.c:286 in `rgbhpg01` (with a coverage of 75.7%) can be parallelized using DVS.

On the other hand, our analysis shows that in most cases either of the three aforementioned sTLP techniques standalone have very low performance potential. For instance, the loop at bmark.c:320 in `a2time01` (with a coverage of 66.2% can be parallelized *iff* all the three types of sTLP

[3]The reader is referred to [26] for an extensive listing of previous work in speculative execution.

	# of Loops		
	Total (w/ vectorization)	# vectorized	Total (w/o vectorization)
a2time01	10	0	10
cjpeg	107	18	98
djpeg	110	16	90
filters	30	0	13
pktflow	19	0	19
bezier01	12	0	12
rotate01	11	0	11
autocor00	17	3	15
conven00	15	0	15
fbital00	6	5	14
fft00	22	0	22
viterb00	16	0	16
qos	27	0	27
tcp	42	3	40

Table 8: Total number of loops before and after vectorization and the number of vectorized loops for applications in EEMBC 1.1 and EEMBC 2.0

techniques are applied. Similarly, parallelization of the loop at rot.c:264 in `rotate01` (with a coverage of 92.8%) would require the application of all the three techniques.

Providing hardware and/or software support for speculative execution can potentially yield better performance. However, this may not be applicable in low-cost embedded systems where the cost and complexity outweigh the performance benefits.

2.3 Impact of Vectorization

Previous studies have shown that vector execution yields high speedups in video and other multimedia applications [27, 28]. In this subsection, we evaluate the impact of SSE-like vectorization on the performance of several applications taken from EEMBC 1.1, 2.0. For this, we determine the difference in loop coverage with and without vectorization. In case of the latter, certain transformations such as loop distribution may not kick in during the optimization phase. Such optimizations are applied (in some cases) to enable vectorization. Consequently, the dynamic loop count of a run with and without vectorization are different (refer to Table 8).

Interestingly, from the table we see that very few loops are actually vectorized by the Intel®compiler.[4] It is important to note that the number of loops vectorized is different from the number of vectorizable loops. To our surprise, we observe that in applications such as `FFT`, no loops are vectorized. This is contrary to the previous studies such as [29], where Franchetti and Püschel showed that FFT is vectorizable. This "anomaly" can be attributed to the specific implementation style of FFT in EEMBC 1.1. For example, let us consider the innermost loop of the FFT computation (taken from `fft00:fft00.c:171`).

```
/* Process butterflies with the same twiddle factors */
for (i = j; i < DataSize; i += n1) {
  l = i + n2;
  tRealData = ( WReal * RealBitRevData[l] ) + ( WImag * ImagBitRevData[l] );
  tImagData = ( WReal * ImagBitRevData[l] ) - ( WImag * RealBitRevData[l] );

/* Scale twiddle products to accomodate 16 bit storage */
  tRealData = tRealData >> BUTTERFLY_SCALE_FACTOR;
  tImagData = tImagData >> BUTTERFLY_SCALE_FACTOR;
  RealBitRevData[l] = RealBitRevData[i] - tRealData;
  ImagBitRevData[l] = ImagBitRevData[i] - tImagData;
  RealBitRevData[i] += tRealData;
  ImagBitRevData[i] += tImagData;
}
```

Clearly, the above loop cannot be vectorized at compile-time due to the unknown value of the variables `n1, n2` and `DataSize` which can potentially lead to a flow dependence [30] between the successive iterations of the loop. The above loop (and such loops in general) can be vectorized at run-time (/dynamically [31]), subject to the values of `n1, n2` and `DataSize`, after applying other transformations such as scalar expansion. However, we find that it is seldom useful to do so due to the high run-time vectorization overhead. This explains why the Intel®compiler did not vectorize the loop. The above limitation can be alleviated by adding pragmas/directives for vectorization, subject to the legality (or correctness) of vectorization of the loop. On the other hand, it can be parallelized with OpenMP-type reduction of the variables `l, tRealData` and `tImagData` and with explicit synchronization. Arguably, one can potentially rewrite the code better or use a different algorithm to enable vectorization, but this is beyond the scope of this paper.

[4]The vector instructions are executed on the MMX units.

	# of Loops	
	Coverage (w/ vectorization)	Coverage (w/o vectorization)
cjpeg	57.3	57.8
djpeg	52.1	53.3
autocor00	82	95.7
fbital00	64.3	65.5
tcp	23.4	24.1

Table 9: Impact of vectorization on loop coverage

In a similar vein, the set of optimizations, their parameters such as the loop unrolling and the unroll factor, and the order in which they are applied is different with vectorization enabled/disabled. This has a direct influence on the loop coverage — it increases in most cases and in some cases it decreases. The decrease can be in part attributed to the limitation of the heuristic which determines the benefit of vectorizing a loop.

The impact of vectorization on the loop coverage of applications with vector loops is shown in Table 9. From the table we see that for each application, except `autocor00`, the reduction in loop coverage (which relates to achievable speedup) due to vectorization is rather small.

This calls for the development of new algorithms, data structures and coding guidelines which are amenable for exploitation SIMD parallelism inherent in embedded applications. Further, profitability analysis techniques need to be developed to balance the trade-off between TLP and SIMD parallelism.

3. CONCLUSIONS

We presented innermost loop-level characterization of industry-standard EEMBC 1.1, 2.0 and the MiBench embedded benchmark suites. It showed that in many programs innermost loops, while perhaps easiest to parallelize, may have low coverage. This is in part due to frequent library function calls which cannot be analyzed by the compiler and to the code in outer loops.

We also evaluated the loop-level speedup achievable via auto-parallelization and simdization. It showed that the speedup achievable via the latter is rather low (as compared to CPU applications [32]). The number of loops parallelized is also be quite low in many applications. This is in part due to the compiler profitability analysis for the target architecture. Small parallel loops cannot be efficiently parallelized even though the task startup overhead is quite modest and are marked serial.

The results indicate that while loop auto-parallelization is a good starting point, it needs a number of additional "improvements". First, parallelism also needs to sought at higher levels such as outer loops with high coverages. However, at such levels auto-parallelization is hard and user assistance, for instance via OpenMP pragmas, is needed. Second, our analysis indicates that libraries have to be further analyzed and optimized to expose parallelism, e.g sequential FFT code in EEMBC is unsuitable for parallel systems. All of these improvements will help in efficient exploitation of the performance potential of the emerging multi-core systems and in meeting the performance requirements of embedded applications.

4. ACKNOWLEDGMENTS

The first author would like to thank Sergey Kozhukhov (Intel, Novosibirsk, Russia) for providing instrumentation support in the Intel compiler used for obtaining the loop coverage. The first author would also like to thank Max Domeika (Intel, Santa Clara, CA) for his help with the installation and build of the EEMBC benchmark suites. In addition, the authors would like to thank Carmen Badea and Babak Salamat for proofreading the paper and for their suggestions.

5. REFERENCES

[1] S. Prakash and A. C. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338–351, 1992.

[2] A. Rae and S. Parameswaran. Application-specific heterogeneous multiprocessor synthesis using differential-evolution. In *Proceedings of the 11th International Symposium on System Synthesis*, pages 83–88, Hsinchu, Taiwan, China, 1998.

[3] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the 38th Design Automation Conference*, pages 518–523, Las Vegas, NV, 2001.

[4] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st Design Automation Conference*, pages 681–685, 2004.

[5] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *Proceedings of the 18th International Conference on VLSI Design*, pages 551–556, Kolkata, India, 2005.

[6] J. T. J. Van Eijndhoven and E. J. D. Pol. Trimedia CPU64 architecture. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 586–592, Austin, Texas, October 1999.

[7] ARM11 Family. `http://www.arm.com/products/CPUs/families/ARM11Family.html`.

[8] Intel® IXP2850 Network Processor. `http://www.intel.com/design/network/products/npfamily/ixp2850.htm`.

[9] OMAP2420. `http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=11990&contentId=4671`.

[10] Intel® Multi-Core Processor Architecture Development. `http://www.intel.com/cd/ids/developer/asmo-na/eng/201969.htm?page=6`.

[11] Dual-Core Intel® Xeon® Processor 7000 sequence Platform Brief. `ftp://download.intel.com/products/processor/xeon/dc7kplatbrief.pdf`.

[12] The Cell Processor. `http://arstechnica.com/articles/paedia/cpu/cell-1.ars`.

[13] RAMP: Research Accelerator for Multiple Processors. `http://ramp.eecs.berkeley.edu/`.

[14] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 58–67, Gold Coast, Australia, May 1992.

[15] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[16] S. Lundstrom and G. Barnes. A controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, St. Charles, IL, August 1980.

[17] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[18] EEMBC. `http://www.eembc.org/`.

[19] MiBench Version 1.0. `http://www.eecs.umich.edu/mibench/`.

[20] OpenMP Specification, version 2.5. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`.

[21] C. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 235–242, August 1987.

[22] R. S. Bird. Notes on recursion elimination. *Communications of the ACM*, 20(6):434–439, 1977.

[23] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 73–82, Boston, MA, 1999.

[24] GNU C library. `http://www.gnu.org/software/libc/`.

[25] R. Gerber, A. J. C. Bik, K. B. Smith, and X. Tian. *The Software Optimization Cookbook, Second Edition.* Intel® Press, 2006.

[26] A. Kejariwal and A. Nicolau. Reading list of performance analysis, speculative execution. `http://www.ics.uci.edu/~akejariw/SpeculativeExecutionReadingList.pdf`.

[27] A. J. C. Bik. *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance.* Intel Press, Hilsboro, OR, 2004.

[28] G. Ren, P. Wu, and D. A. Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, page 89b, Denver, CO, 2005.

[29] F. Franchetti and M. Puschel. Short vector code generation for the discrete fourier transform. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 58, Nice, France, 2003.

[30] D. Kuck. *The Structure of Computers and Computations, VOLUME 1.* John Wiley and Sons, New York, NY, 1978.

[31] S. Vajapeyam, P. J. Joseph, and T. Mitra. Dynamic vectorization: A mechanism for exploiting far-flung ILP in ordinary programs. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 16–27, Atlanta, GA, 1999.

[32] SPEC: Standard Performance Evaluation Corporation. `http://www.spec.org/`.