Towards Automatic Profile-Driven Parallelization of Embedded Multimedia Applications

Georgios Tournavitis and Björn Franke

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh
United Kingdom

Abstract. Despite the availability of ample parallelism in multimedia applications parallelizing compilers are largely unable to extract this application parallelism and map it onto existing embedded multi-core platforms. This is mainly due to the limitations of traditional auto-parallelization on static analysis and loop-level parallelism. In this paper we propose a dynamic, profile-driven approach to auto-parallelization targeting coarse-grain parallelism. We present our methodology and tools for the extraction of task graphs from sequential codes and demonstrate the various stages involved in this process based on the JPEG-2000 still image compression application. In addition, we show how the joint detection of multiple levels of parallelism and exploitation of application scenarios can lead to performance levels close to those obtained by manual parallelization. Finally, we demonstrate the applicability of our methodology to a broader set of embedded multimedia codes.

1 Introduction

In recent years multi-core computing systems have become widely available and their efficiency in multimedia, signal processing, wireless or graphic applications has been already proved [1, 2]. However, automatic mapping of applications onto these platforms is still a major research challenge. The predominant manual approach to application parallelization is clearly not scalable enough to cope with the growing complexity of embedded codes and is bound to fail without strong programming tools support.

While auto-parallelization has a long research history, success in the embedded systems domain is limited. This is especially true for embedded multimedia applications which exhibit properties and constraints significantly different from the traditional scientific high-performance computing domain targeted by auto-parallelizers such as e.g. SUIF [3], Polaris [4]. In order to guarantee correctness parallelizing compilers heavily depend on static analysis for data dependence testing.

In this paper we propose to incorporate additional dynamic profiling information in the extraction of coarse-grain task graphs. We address the large body

of legacy codes written in sequential programming languages like C and present a pragmatic approach to parallelization combining static and profile-driven analysis as well as multiple levels of parallelism. Based on the JPEG-2000 still image compression application we demonstrate how to automatically derive a parallel OpenMP implementation and compare its performance with both a manually parallelized implementation and one generated by a static state-of-the-art auto-parallelizing compiler.

Contributions The recent advent of multi-core architectures has sparked interest in automatic extraction of coarse-grain parallelism from sequential programs, e.g. [2, 5, 6]. We extend it in numerous ways and among the specific contributions of our paper are:

- the introduction of an *"executable IR"* that simplifies the back-annotation of high-level code and data structures in the compiler with information collected during profiling,
- the joint extraction of task graphs and the detection of application scenarios [7] in order to efficiently exploit the semi-dynamic nature of many multimedia applications,
- the automatic detection of scalar and array reduction statements, using a *hybrid* statical and profile-driven analysis.

Overview This paper is structured as follows. In section 2 we present our parallelization methodology and tools. This is followed by a case study based on the JPEG-2000 application in section 3 before we evaluate our parallelization approach on a wider range of embedded multimedia applications in 4. We discuss related work in section 5 and summarize and conclude in section 6.

2 Profiling toolchain

At the heart of the proposed parallelization methodology is our profile-driven dependence analyzer. In the following paragraphs we give a brief technical description of the main components of our current prototype. An overview of the compilation, profiling and analysis process is depicted in Figure 1.

2.1 Source Code Instrumentation

Our primary objective is to enhance and disambiguate the static structure of a given program using precise, dynamic information. The main obstacle to this process is correlating the binary program execution back to the source code. Unfortunately, traditional compiler flows discard most of the high-level information and source-level constructs during the code generation process. Thus, binary instrumentation approaches require the use of debugging information which are imprecise since only symbol table information and the corresponding line-of-code of an instruction is maintained.
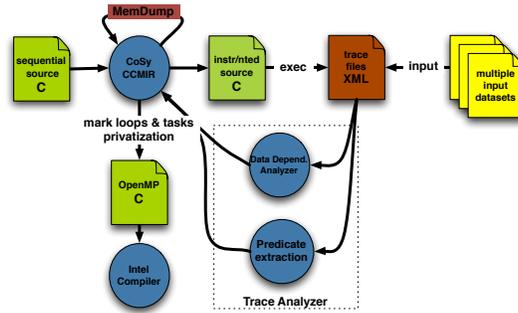
Fig. 1. Overview of the compilation, profiling and parallelization flow.

In contrast, our instrumentation framework (*MemDump*) operates on a medium-level Intermediate Representation (IR) of the CoSy compiler and can emit a plain C representation of its IR, enabling fast, native execution. Effectively we create an *"executable IR"* that allows for an efficient and precise *back-annotation* of profiling information to the compiler internal data structures such as data dependence graphs. This differs e.g. from [6] where an instrumented microarchitectural simulator is used for profiling and difficulties arising from the particular idiosyncrasies of the underlying instruction set architecture have to be addressed explicitly.

2.2 Profile-Driven Dependence Analyzer

The profile analyzer operates on the previously generated traces and evaluates memory and control-flow operations to dynamically construct data and control flow representations of the program under inspection. While a large number of executed IR nodes need to be processed this process is fast as no low-level machine details need to be considered.

Trace parsing Each "trace item" is processed using the procedure described in Algorithm 1. During this process, the control-flow handler reconstructs a global Control Flow Graph (CFG) of the application and maintains context information (e.g. call and loop stack, normalized loop-iteration vector). The corresponding data-flow handler keeps a mapping from the memory address space of the application to the nodes of the CFG. For each memory location it records the node which modified (def) it last and the normalized iteration vector at that time. In the current prototype the memory granularity is at a byte-level. Data-dependence information is registered in the form of *data-edges* which are inserted in an overlay of the CFG. Each data-edge is annotated with information about the address regions that are communicated from the source to the target node, creating a Control Data Flow Graph (CDFG) of the application. Finally, each data edge contains a bit-vector which records on which levels of the loop-nest this particular edge was carrying a loop-carried dependence.

Algorithm 1: The main algorithm executed by the Profile-Driven Dependence Analyzer to extract the CDFG.

Data
· $CDFG(V, E_C, E_D)$: graph with control (E_C) and data-flow (E_D) edges
· $bit_e[]$: bitfield in each $e \in E_D$
· set_e: address set in each $e \in E_D$
· $it_a[]$: iteration vector of address a
· $M[A, \{V, it\}]$: hastable maps memory addresses $\rightarrow \{V, it_a\}$ tuple
· $it_0[]$: current normalized iteration vector
· $u \in V$: current node

Procedure *instruction_handler*
$I \leftarrow$ next instruction from trace
if *I is a memory instruction* **then**
    $a \leftarrow$ address accessed by instruction
    if *I is a DEF* **then**
        update last writer in M
    else if *USE* **then**
        find matching def from M
        if *def$\rightarrow$use edge $e \notin$ CDFG* **then**
            add e in E_D
        $set_e \leftarrow set_e \cup \{a\}$
        foreach $i : it_a[i] \neq it_0[i]$ **do** $bit_e[i] \leftarrow true$
        $it_a \leftarrow it_0$

else if *I is a control instruction* **then**
    $v \leftarrow$ node referenced by instruction
    if *edge $(u, v) \notin E_C$* **then**
        add (u, v) in $CDFG$
    $u \leftarrow v$

Privatization We maintain a complete list of true-, anti- and output-dependencies as these are required for parallelization. Rather than recording all the readers of each memory location we keep a map of the normalized iteration index of each memory location that is read/written at each level of a loop-nest. This allows us to efficiently track all memory locations that cause a loop-carried anti- or output-dependence. A scalar x is privatizable within a loop if and only if every path from the beginning to the loop body to a use of x passes from a definition of x before the use. Hence, we can determine the privatizable variables by inspecting the incoming and outgoing data-dependence edges of the loop.

Parallelism detection After the trace processing has been completed, the analyzer performs an additional post-analysis based on the augmented CFG to determine parallelizable loops and control-independent basic-block regions. In addition, for each loop the analyzer produces a detailed summary of variables that have to be privatized to eliminate any false-dependencies. Finally, whenever a loop is not parallelizable we inform the user about the offending data-dependencies, providing information about the (i) source/destination node, (ii) the data which was communicated and (iii) the context it occurred.

Reduction detection In the presence of pointer or indirect array accesses (e.g. sparse matrices) static analysis makes conservative assumptions and, thus, limits the amount of exploitable parallelism. We enhance the existing static analysis and propose a *hybrid approach* for these complex cases. As a first step we use a simple static analysis pass in our compiler to select reduction statement *candi-*

dates. Then, we instrument our source code to explicitly tag these operations. Finally, the dependence analyzer determines which candidates are *valid* reductions. The reduction detection step is based on the following necessary properties of a reduction:

1. The reduction operator is both commutative and associative.
2. There is a true self-dependence which denotes the accumulation to the partial result of the reduction.
3. Only the final result of the reduction is used later i.e there is no outgoing dependency from the reduction.
4. No true dependency is fed into the reduction.

It is important to stress that this methodology not only tracks scalar and array reductions but also *coupled* reductions, where more than one statements of the loop "accumulate" on the same targets.

2.3 Parallelization Methodology

The main stages of our parallelization approach are:

1. Source code instrumentation using *MemDump*
2. Compilation of the instrumented source using a standard ANSI-C compiler.
3. Program execution and generation of multiple trace files for a set of representative input datasets.
4. Processing of the trace files with the profile-driven dependence analyzer.
5. Identification of application hot spots.
6. Parallelization of the hotspots based on dynamic dependence information, starting with the outer-most loops.
7. Parallel code generation for the selected regions using one of the following OpenMP directives.
   - *Parallel for* for well structured parallelizable *for*-loops.
   - *Parallel taskq* for any uncountable or not well structured parallelizable loops[1].
   - *Parallel sections* for independent coarse-grain functions or control-independent code-regions.
8. Guarding of loop-carried dependencies either using *atomic*, *ordered* or *reduction* OpenMP clauses.
9. Privatization of variables that according to the dependence analyzer are causing a false-dependence.

---

[1] Task queues are an extension to the OpenMP standard already available in the Intel compilers, but also part of the recently-released OpenMP 3.0 standard.

2.4 Scenarios and Predicate Extraction

Application scenarios [7] correspond to "modes of operation" of an application and can be used to switch between individually optimized code versions, or prefetch data from memory or optimize communication if a suitable predictor can be constructed.

For sufficiently large sub-regions of the CFG we analyse the conditional statements at their roots and trace the variables involved in the condition expressions back to their original definitions. At this point we construct a new predicate expression that can be used as an early indicator about the application's future control flow. For many multimedia applications, for example, we have found that the statically undecidable control flow can be determined as soon as the relevant header information has been read from the input stream, i.e. long before the actual branch to a particular code region (e.g. a specific mode of a multimedia codec) is taken.

2.5 Complexity and applicability of the approach

As we process data dependence information at byte-level granularity we may need to maintain data structures growing potentially as large as the entire address space of the target platform. In practice, however, we have not observed any cases where more than 1GB of heap memory was needed to maintain the dynamic data dependence structures, even for applications taken from the SPEC 2000 integer and floating-point benchmark suites.

While the dynamic traces can reach several GB's in total size, they can be processed online. Our tools are designed to build their data structures incrementally as the input trace is streamed into the analyser, thus eliminating the need for large traces to be stored.

As our tools operate on the same level as the medium-level compiler IR we do not need to consider detailed architecture state, hence our tools can operate very efficiently. In fact, we only need to keep track of memory and control flow operations and make incremental updates to hash tables and graph structures. In practice, we have observed processing times ranging from a few seconds for small kernel benchmarks up to about 60mins for large multimedia codecs operating on long video streams.

2.6 Safety

Despite the fact that our method cannot guarantee correctness and still presumes user verification, in all the benchmarks that we considered so far there was not a single case of a loop/region falsely identified as parallel (*false positive*). In addition, the high potential of thread-level parallelism and the immense performance gap between automatic and manual parallelization, urge for a more pragmatic approach. We consider the tools and the synergistic techniques proposed in this paper to be a step towards this direction rather than a replacement of static dependence analysis.

3 Case Study: JPEG-2000 Still Image Compression

In this section we present a case study based on the state-of-the-art JPEG-2000 image coding standard. We explain our parallelization approach for an open-source implementation of the standard, OpenJPEG, and report performance figures for the resulting OpenMP implementation.

3.1 JPEG-2000 Still Image Compression

JPEG-2000 is the most recent standard produced by the JPEG committee for still image compression and is widely used in digital photography, medical imaging and the digital film industry. The main stages of the wavelet-based coding process as defined in the JPEG-2000 standard are shown in the diagram in figure 2.
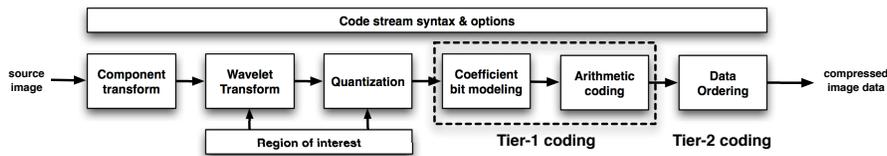

Fig. 2. Overview of the JPEG-2000 coding process.

3.2 Detection of Parallelism

JPEG-2000 exhibits multiple levels of data-parallelism (e.g. tiles, components, code-blocks). Even without tiling most of the coding stages can be performed independently on each component. Furthermore, during entropy bit coding, an additional level of parallelism can be exploited using a code-block decomposition. In the following paragraphs we go over the individual steps involved in the parallelization according to the methodology presented in section 2.2.

Static Analysis of DOALL Loops Initially, we evaluated the amount and type of parallelism a production auto-parallelizing compiler (Intel *icc* v10.1) can extract from our application. Enabling all optimizations available in the Intel compiler (e.g. multi-file interprocedural analysis, auto-parallelization and auto-vectorization) had no impact on the overall performance of the application for both compression and decompression. The compiler vectorized 17 and 11 loops, respectively, but could not parallelize any loops. Setting the profitability threshold to its minimum value resulted in 31 and 20 vectorized loops for compression and decompression, respectively, and 54 and 27 parallelized loops. This more aggressive parallelization scheme, however, resulted in a slow-down of 8 over the default setting.

Profile-Driven Analysis Following hot spot detection we use our analysis tool described in section 2 to extract parallelism at any level of the call hierarchy under the detected hot spots *dwt_encode* and *t1_encode_cblks* of the JPEG-2000 application.

While the main loop of the discrete wavelet transform can be easily expressed as a parallel DOALL loop, the situation for *t1_encode_cblks* is more complicated. Two loop-carried dependencies at the outermost loop level prevent parallelization at this level. Using the automatic reduction detection of the Trace Analyzer, we can automatically detect that this update can be substituted with an appropriate parallel reduction operation. Indeed, manual code inspection revealed that this is due to the update of an accumulation variable which is not used before later stages of the encoding process.

Another issue arises from the particular dynamic memory allocation and buffer management scheme which was used in the function *t1_encode_cblks* to minimize the cost of repetitive calls to *malloc()*. The particular buffer allocation function causes another loop-carried dependence detected by our tools. After privatization of the temporary buffer each thread manages its own buffer, eliminating the previously detected dependence. However, at this point our tool still requires manual intervention as it cannot be inferred that calls to the library function *malloc()* can be reordered. In an improved version of our tools we will include this information for this and other system functions.

3.3   Deriving a Parallel Implementation

After we have identified the available parallelism, the next step is to actually derive a parallel OpenMP implementation.

While OpenMP eliminates much the burden of explicit thread synchronization, we still need to address data-sharing. Basically, there are two kinds of memory access, *shared* and *private*. Based on the true data dependencies we classify the variables accordingly and for each private variable a separate copy is created for each thread.

Dynamic data structures need special attention to preserve correctness. In the case of *dwt_encode* it is straightforward for our tool to privatize the dynamically allocated data structure, since these are allocated within the dynamic context of the function. On the other hand, for *t1_encode_cblks* the situation more complicate since the allocation is just before the call to the function. Therefore, we need to ensure that the allocation is performed independently for each thread. We achieve this by propagating the parallel region out of the scope of *t1_encode_cblks* up to the point where the allocation takes place. The work-sharing directive will still result in the same parallel loop execution, but accesses to shared data before the work-sharing construct are executed only by the original thread in a region protected by an OpenMP *master* directive.

3.4 Performance Evaluation

We evaluated the achievable speedup of the parallelized benchmark on an SMP workstation ($2 \times$ Intel Dual-Core Xeon $5160/3GHz$), running Fedora Core 6 Linux. Both the sequential and OpenMP versions of the code were compiled using Intel® Compiler (v10.1) and optimization level $-O3$. The reported speedups are over the unmodified sequential code.

Performance results of both encoding and decoding using either parallelization at component or block-level of *t1_encode_cblks* are shown in figure 3. Given that we targeted approximately 70% of the original, sequential execution time in our parallelization and the theoretical limit given by Amdahl's Law, obtained speedups of $\approx 2\times$ are clearly promising. Comparing the performance of the component-level and the block-level decomposition it is clear that the latter is more scalable since the maximum number of components in an image is three. This also explains why the performance on four cores in Figure 3 is almost identical to the one on three cores.
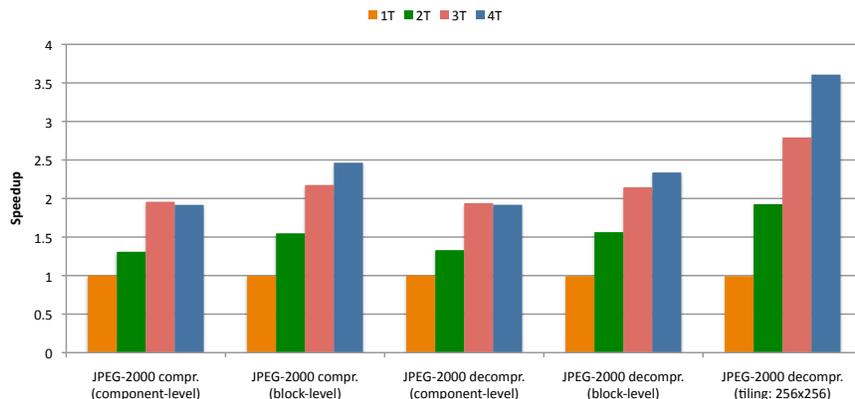

Fig. 3. Speedups achieved for various functional modes of the JPEG-2000 codec.

Tiling Parallelizing JPEG-2000 decompression at a tile-level achieved far better scalability than the one-tile parallelization schemes, reaching a speedup $3.6\times$ on four cores and 256×256 tiles. In fact, these results are in line with those obtained by manual parallelization by expert programmers [9, 10]

Scenarios For the JPEG-2000 application we have extracted execution scenarios and computed predicate expressions that can be used to determine later tasks at the earliest possible point in the program execution. The main observation is that this input-dependent behavior is in fact observable at the control-flow level. Typically, the cause for variability in multimedia applications is restricted to a small set of input parameters (e.g. file header information or command line arguments). The parameters that we experiment with were the following (A) turning lossy compression true/false, (ii) using one or multiple tiles of various

sizes qnd (*iii*) using source images of one or three components. In all cases the predicates generated by our tools accurately describe the precise comparisons that distinguish the four possible states. After finding these predicates, the next step is to locate the earliest point of the program where these predicates can be safely evaluated. As expected, this is the moment the relevant fields are read from file header.

4 Broader Evaluation

In this section we present additional performance results for selected applications taken from the Mediabench, UTDSP and MiBench benchmark suites. We have selected these applications based on their suitability for parallelization and compatibility with our tools.

In Figure 4 we present the speedups for eleven parallelized benchmarks. The average speedup using 4 threads is 2.66. Applications with ample coarse-grain DOALL parallelism in outer loops like *susan*, *stringsearch* and *compress* achieve relatively higher speedups and present good scalability. On the other hand applications, like *histogram*, and *lpc* exhibit either large sequential parts, or are only parallelizable in lower-levels of nested-loops. As a consequence, these applications are not scaling so well, attaining relatively lower speedups. In addition, application like *FFT* and *epic* manage to get significant performance gains despite the considerable slowdowns (57% and 32% respectively) of the single-threaded execution of the parallel code.
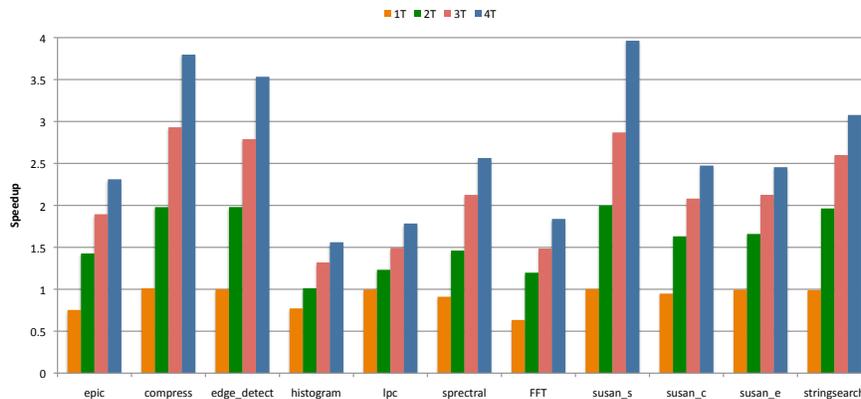

Fig. 4. Speedups achieved using the parallelized implementations of the benchmarks on a 4-core machine.

5 Related Work

Dynamic Dependence Analysis (DDA) [11] is a method to extract dependence information using the actual data accesses recorded during the runtime of a

program execution and aims at improving loop-level parallelization of scientific applications. Hybrid dependence analysis [12] unifies static and run-time analysis and has been successfully applied to automatic parallelization, where it helped extracting more loop-level parallelism in scientific codes.

Similarly, in [13] DDA is investigated as an aggressive, i.e. unsafe, dependence checking method to uncover loop-level parallelism. They present a study of the parallelization potential previously unexploited by the SUIF parallelizing compiler for a set of Digital Signal Processing (DSP) kernels and multimedia applications. The same authors show in [14] and [15] that dependence vectors extracted by means of DDA can be exploited to combine loop parallelization and functional pipelining. This work, however, is focused on the general feasibility of using dynamic dependence information, but does not present a specific methodology for deriving a parallel implementation.

The techniques presented in [6] are most relevant to our work. Load and stores memory operations are instrumented in Dynamic SimpleScalar to extract information about the data producer and consumer functions of the program. Based on this profile information an interprocedural data flow graph is constructed. This graph and additional profiling information summarizing the execution time of each function are used to cluster functions to form a functional pipeline. Then, this information is provided to the user who decides and implements the parallel code using multithreading. In a case study this method is evaluated against the *bzip2* benchmark, but it lacks features like code generation, automatic detection of higher-level parallelization constructs (e.g. reductions) and incorporation of the statically available information of our approach.

The problem of pipeline-parallelism exploitation in C programs is addressed in [16]. Based on user directed source code annotations a task pipeline is constructed and communication across task boundaries is tracked by means of dynamic analysis. While this approach is practical, it presumes programmer's insight to the underlying algorithm and it requires extensive user guidance and manual intervention.

MAPS [5] uses profiling data to annotate CDFGs for the extraction of task graphs from sequential codes. This technique is heavily dependent on a code partitioning framework (TCT) which targets a specific distributed memory MPSoC and assumes accurate static data dependence analysis. In addition, this approach has only been evaluated against two applications, JPEG and ADPCM.

Dependence-profiling approaches are also used in the context of Thread-Level Speculation (TLS) [17–19]. Unlike our proposal which uncovers *always-true* parallelism, TLS approaches use profiling information to mitigate the cost due to mispeculation on *may-dependencies* while preserving the sequential semantics.

6   Conclusions

We have demonstrated that profile-driven parallelization is a powerful methodology to uncover parallelism from complex applications such as JPEG-2000 and other media processing codes. Our automated approach to parallelism extrac-

tion and OpenMP code generation is capable of delivering performance figures competitive to those achieved by expert programmers during manual parallelization. We have shown that we can derive alternative parallelization schemes corresponding to different parallelization granularities, handle dynamically allocated data structures and compute predicate expressions for observed application scenarios.

References

1. Stolberg, H.J., Berekovic, M., et al.: HiBRID-SoC: a multi-core system-on-chip architecture for multimedia signal processing applications. DATE (2003)
2. Takamichi, M., Saori, A., et al.: Automatic parallelization for multimedia applications on multicore processors. IPSJ SIG Technical Reports **4** (2007)
3. Hall, M.W., Anderson, J.M., et al.: Maximizing multiprocessor performance with the SUIF compiler. Computer **29**(12) (1996)
4. Blume, W., Eigenmann, R., et al.: Effective automatic parallelization with PO-LARIS. IJPP (Jan 1995)
5. Ceng, J., Castrillon, J., et al.: MAPS: an integrated framework for mpsoc application parallelization. DAC (2008)
6. Rul, S., Vandierendonck, H., et al.: Function level parallelism driven by data dependencies. ACM SIGARCH Computer Architecture News **35**(1) (3 2007)
7. Gheorghita, S., Palkovic, et al. A system scenario based approach to dynamic embedded systems. ToDAES to appear.
8. Marcellin, M.W., Gormish, M.J.: The JPEG-2000 Standard. Kluwer Academic Publishers, Norwell, MA (1995)
9. Meerwald, P., Norcen, R., et al.: Parallel JPEG2000 image coding on multiprocessors. IPDPS (2002)
10. Norcen, R., Uhl, A.: High performance JPEG 2000 and MPEG-4 VTC on SMPs using OpenMP. Parallel Computing (2005)
11. Peterson, P., Padua, D.P.: Dynamic dependence analysis: A novel method for data dependence evaluation. LCPC (1992)
12. Rus, S., Rauchwerger, L.: Hybrid dependence analysis for automatic parallelization. Technical report (2005)
13. Karkowski, I., Corporaal, H.: Overcoming the limitations of the traditional loop parallelization. HPCN (1997)
14. Karkowski, I., Corporaal, H.: Design of heterogenous multi-processor embedded systems: Applying functional pipelining. IEEE PACT (1997)
15. Karkowski, I., Corporaal, H.: Exploiting fine- and coarse-grain parallelism in embedded programs. IEEE PACT (1998)
16. Thies, W., Chandrasekhar, V., et al.: A practical approach to exploiting coarse-grained pipeline parallelism in C programs. MICRO (2007)
17. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: Posh: a tls compiler that exploits program structure. PPoPP (2006)
18. Carlos García Qui n., Madriles, C., et al.: Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. PLDI (2005)
19. Wu, P., Kejariwal, A., Cascaval, C.: Compiler-driven dependence profiling to guide program parallelization. LCPC (2008)