Towards a Holistic Approach to Auto-Parallelization

Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping

Georgios Tournavitis Zheng Wang Björn Franke Michael F.P. O'Boyle

Institute for Computing Systems Architecture (ICSA)
School of Informatics
University of Edinburgh
Scotland, United Kingdom
gtournav@inf.ed.ac.uk,jason.wangz@ed.ac.uk,{bfranke,mob}@inf.ed.ac.uk

Abstract

Compiler-based auto-parallelization is a much studied area, yet has still not found wide-spread application. This is largely due to the poor exploitation of application parallelism, subsequently resulting in performance levels far below those which a skilled expert programmer could achieve. We have identified two weaknesses in traditional parallelizing compilers and propose a novel, integrated approach, resulting in significant performance improvements of the generated parallel code. Using profile-driven parallelism detection we overcome the limitations of static analysis, enabling us to identify more application parallelism and only rely on the user for final approval. In addition, we replace the traditional target-specific and inflexible mapping heuristics with a machine-learning based prediction mechanism, resulting in better mapping decisions while providing more scope for adaptation to different target architectures. We have evaluated our parallelization strategy against the NAS and SPEC OMP benchmarks and two different multi-core platforms (dual quad-core Intel Xeon SMP and dual-socket QS20 Cell blade). We demonstrate that our approach not only yields significant improvements when compared with state-of-the-art parallelizing compilers, but comes close to and sometimes exceeds the performance of manually parallelized codes. On average, our methodology achieves 96% of the performance of the hand-tuned OpenMP NAS and SPEC parallel benchmarks on the Intel Xeon platform and gains a significant speedup for the IBM Cell platform, demonstrating the potential of profile-guided and machine-learning based parallelization for complex multi-core platforms.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Experimentation, Languages, Measurement, Performance

Keywords Auto-Parallelization, Profile-Driven Parallelism Detection, Machine-Learning Based Parallelism Mapping, OpenMP

PLDI'09, June 15–20, 2009, Dublin, Ireland.

1. Introduction

Multi-core computing systems are widely seen as the most viable means of delivering performance with increasing transistor densities (1). However, this potential cannot be realized unless the application has been well parallelized. Unfortunately, efficient parallelization of a sequential program is a challenging and error-prone task. It is generally agreed that manual code parallelization by expert programmers results in the most streamlined parallel implementation, but at the same time this is the most costly and time-consuming approach. Parallelizing compiler technology, on the other hand, has the potential to greatly reduce cost and time-to-market while ensuring formal correctness of the resulting parallel code.

Automatic parallelism extraction is certainly not a new research area (2). Progress was achieved in 1980s to 1990s on restricted *DOALL* and *DOACROSS* loops (3; 4; 5). In fact, this research has resulted in a whole range of parallelizing research compilers, e.g. Polaris (6), SUIF-1 (7) and, more recently, Open64 (8). Complementary to the on-going work in auto-parallelization many high-level parallel programming languages – such as Cilk-5 (9), OpenMP, StreamIt (10), UPC (11) and X10 (12) – and programming models – such as Galois (14), STAPL (15) and HTA (16) – have been proposed. Interactive parallelization tools (17; 18; 19; 20) provide a way to actively involve the programmer in the detection and mapping of application parallelism, but still demand great effort from the user. While these approaches make parallelism expression easier than in the past, the effort involved in discovering and mapping parallelism is still far greater than that of writing an equivalent sequential program.

This paper argues that the lack of success in auto-parallelization has occurred for two reasons. First, traditional static parallelism detection techniques are not effective in finding parallelism due to lack of information in the static source code. Second, no existing integrated approach has successfully brought together automatic parallelism discovery and portable mapping. Given that the number and type of processors of a parallel system is likely to change from one generation to the next, finding the right mapping for an application may have to be repeated many times throughout an application's lifetime, hence, making automatic approaches attractive.

Approach. Our approach integrates profile-driven parallelism detection and machine-learning based mapping in a single framework. We use profiling data to extract actual control and data dependences and enhance the corresponding static analyses with dynamic information. Subsequently, we apply a previously trained

```
for (i = 0; i < nodes; i++) {
  Anext = Aindex[i];
  Alast = Aindex[i + 1];

  sum0 = A[Anext][0][0]*v[i][0] +
         A[Anext][0][1]*v[i][1] +
         A[Anext][0][2]*v[i][2];
  sum1 = ...

  Anext++;
  while (Anext < Alast) {
    col = Acol[Anext];

    sum0 += A[Anext][0][0]*v[col][0] +
            A[Anext][0][1]*v[col][1] +
            A[Anext][0][2]*v[col][2];
    sum1 += ...

    w[col][0] += A[Anext][0][0]*v[i][0] +
                 A[Anext][1][0]*v[i][1] +
                 A[Anext][2][0]*v[i][2];
    w[col][1] += ...
    Anext++;
  }
  w[i][0] += sum0;
  w[i][1] += ...
}
```

Figure 1. Static analysis is challenged by sparse array reduction operations and the inner *while* loop in the SPEC *equake* benchmark.

machine-learning based prediction mechanism to each parallel loop candidate and decide if and how the parallel mapping should be performed. Finally, we generate parallel code using standard OpenMP annotations. Our approach is semi-automated, i.e. we only expect the user to finally approve those loops where parallelization is likely to be beneficial, but correctness cannot be proven conclusively.

Results. We have evaluated our parallelization strategy against the NAS and SPEC OMP benchmarks and two different multi-core platforms (dual quad-core Intel Xeon SMP and dual-socket QS20 Cell blade). We demonstrate that our approach not only yields significant improvements when compared with state-of-the-art parallelizing compilers, but comes close to and sometimes exceeds the performance of manually parallelized codes. We show that profiling-driven analyses can detect more parallel loops than static techniques. A surprising result is that all loops classified as parallel by our technique are correctly identified as such, despite the fact that only a single, small data input is considered for parallelism detection. Furthermore, we show that parallelism detection in isolation is not sufficient to achieve high performance, and neither are conventional mapping heuristics. Our machine-learning based mapping approach provides the adaptivity across platforms that is required for a genuinely portable parallelization strategy. On average, our methodology achieves 96% of the performance of the hand-tuned OpenMP NAS and SPEC parallel benchmarks on the Intel Xeon platform, and a significant speedup for the Cell platform, demonstrating the potential of profile-guided machine-learning based auto-parallelization for complex multi-core platforms.

Overview. The remainder of this paper is structured as follows. We motivate our work based on simple examples in section 2. This is followed by a presentation of our parallelization framework in section 3. Our experimental methodology and results are discussed in sections 4 and 5, respectively. We establish a wider context of related work in section 6 before we summarize and conclude in section 7.

```
#pragma omp for reduction(+:sum) private(d)
for (j=1; j <= lastcol-firstcol-1; j++) {
    d = x[j] - r[j];
    sum = sum + d * d;
}
```

Figure 2. Despite its simplicity mapping of this parallel loop taken from the NAS *cg* benchmark is non-trivial and the best-performing scheme varies across platforms.

2. Motivation

Parallelism Detection. Figure 1 shows a short excerpt of the *smvp* function from the SPEC *equake* seismic wave propagation benchmark. This function implements a general-purpose sparse matrix-vector product and takes up more than 60% of the total execution time of the *equake* application. While conservative, static analysis fails to parallelize both loops due to sparse matrix operations with indirect array indices and the inner *while* loop, profiling-based dependence analysis provides us with the additional information that no actual data dependence inhibits parallelization for a given sample input. While we still cannot prove absence of data dependences for *every possible* input we can classify both loops as candidates for parallelization (reduction) and, if profitably parallelizable, present it to the user for approval. In this example, the user would provide the additional knowledge (and guarantee) that every `col` index in the inner loop is unique and, hence, accesses to `w[col][0]` and `w[col][1]`, respectively, do not result in cross-iteration dependencies.

This example demonstrates that static analysis is overly conservative. Profiling based analysis, on the other hand, can provide accurate dependence information for a *specific* input. When combined we can select candidates for parallelization based on *empirical evidence* and, hence, can eventually extract more application parallelism than purely static approaches.

Mapping. In figure 2 a parallel reduction loop originating from the parallel NAS conjugate-gradient *cg* benchmark is shown. Despite the simplicity of the code, mapping decisions are non-trivial. For example, parallel execution of this loop is not profitable for the Cell BE platform due to high communication costs between processing elements. In fact, parallel execution results in a massive slowdown over the sequential version for the Cell for any number of threads. On the Intel Xeon platform, however, parallelization can be profitable, but this depends strongly on the specific OpenMP scheduling policy. The best scheme (*STATIC*) results in a speedup of 2.3 over the sequential code and performs 115 times better than the worst scheme (*DYNAMIC*) that slows the program down to 2% of its original, sequential performance.

This example illustrates that selecting the correct mapping scheme has a significant impact on performance. However, the mapping scheme varies not only from program to program, but also from architecture to architecture. Therefore, we need an automatic and portable solution for parallelism mapping.

3. Parallelization Framework

In this section we provide an overview and technical details of our parallelization framework.

As shown in figure 3, a sequential C program is initially extended with plain OpenMP annotations for parallel loops and reductions as a result of our profiling-based dependence analysis. In

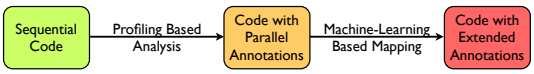


Figure 3. A two-staged parallelization approach combining profiling-driven parallelism detection and machine-learning based mapping to generate OpenMP annotated parallel programs.

addition, data scoping for shared and private data also takes place at this stage.

In a second step we add further OpenMP work allocation clauses to the code if the loop is predicted to benefit from parallelization, or otherwise remove the parallel annotations. This also happens for loop candidates where correctness cannot be proven conclusively (based on static analysis) and the user disapproves of the suggested parallelization decision.

Finally, the parallel code is compiled with a native OpenMP compiler for the target platform. A complete overview of our toolchain is shown in figure 4.

3.1 Profile-Driven Parallelism Detection

We propose a profile-driven approach to parallelism detection where the traditional static compiler analyses are not replaced, but *enhanced* with dynamic information. To achieve this we have devised a novel instrumentation scheme *operating at the intermediate representation (IR) level* of the compiler. Unlike e.g. (21) we do not need to deal with low-level artifacts of any particular instruction set, but obtain dynamic control and data flow information relating to IR nodes immediately. This allows us to *back-annotate* the original IR with the profiling information and resume compilation/parallelization. The three stages involved in parallelism detection are:

1. IR instrumentation, C code generation and profiling
2. CDFG construction and dependence analysis
3. Parallel code generation

3.1.1 Instrumentation and Profile Generation

Our primary objective is to enhance the static analysis of a traditional parallelizing compiler using precise, dynamic information. The main obstacle here is correlating the low-level information gathered during program execution – such as specific memory accesses and branch operations – to the high-level data and control flow information. Debug information embedded in the executable is usually not detailed enough to enable this reconstruction.

To bridge this *information gap* we perform instrumentation at the IR level of the compiler (CoSy). For each variable access, additional code is inserted that emits the associated symbol table reference as well as the actual memory address of the data item. All data items including arrays, structures, unions and pointers are covered in the instrumentation. This information is later used to disambiguate memory accesses that static analysis fails to analyze. Similarly, we instrument every control flow instruction with the IR node identifier and code to record the actual, dynamic control flow. Eventually, a plain C representation close to the original program, but with additional instrumentation code inserted, is recovered using an IR-to-C translation pass and compiled with a native x86 compiler.

The program resulting from this process is still sequential and functionally equivalent to the original code, but emits an additional trace of data access and control flow items.

3.1.2 CDFG Construction and Dependence Analysis

The subsequent analysis stage consumes one trace item at a time and incrementally constructs a global control and data flow graph (CDFG) on which the parallelism detection is performed. Hence, it is not necessary to store the entire trace if the tools are chained up appropriately.

Each trace item is processed by algorithm 1. It distinguishes between control and data flow items and maintains various data structures supporting dependence analysis. The control flow section constructs a global control flow graph of the application including call stacks, loop nest trees, and normalized loop iteration vectors. The data-flow section is responsible for mapping memory addresses to specific high-level data flow information. For this we keep a hash table where data items are traced at byte-level granularity. Data dependences are recorded as data edges in the CDFG. These edges are further annotated with the specific data sections (e.g. array indices) that cause the dependence. For loop-carried data dependences an additional bit vector relating the dependence to the surrounding loop nest is maintained.

Data Items
· $CDFG(V, E_C, E_D)$: 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
· $M[A, \{V, it\}]$: hash table: mem. addr. $\rightarrow \{V, it_a\}$
· $it_0[]$: current normalized iteration vector
· $u \in V$: current node

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

Algorithm 1: Algorithm for CDFG construction.

As soon as the complete trace has been processed the constructed CDFG with all its associated annotations is imported back into the CoSy compiler and added to the internal, statically derived data and control flow structures. This is only possible because the dynamic profile contains references to IR symbols and nodes in addition to actual memory addresses.

The profiling-based CDFG is the basis for the further detection of parallelism. However, there is the possibility of conflicting dependence information, for example, if a *"may"* data dependence

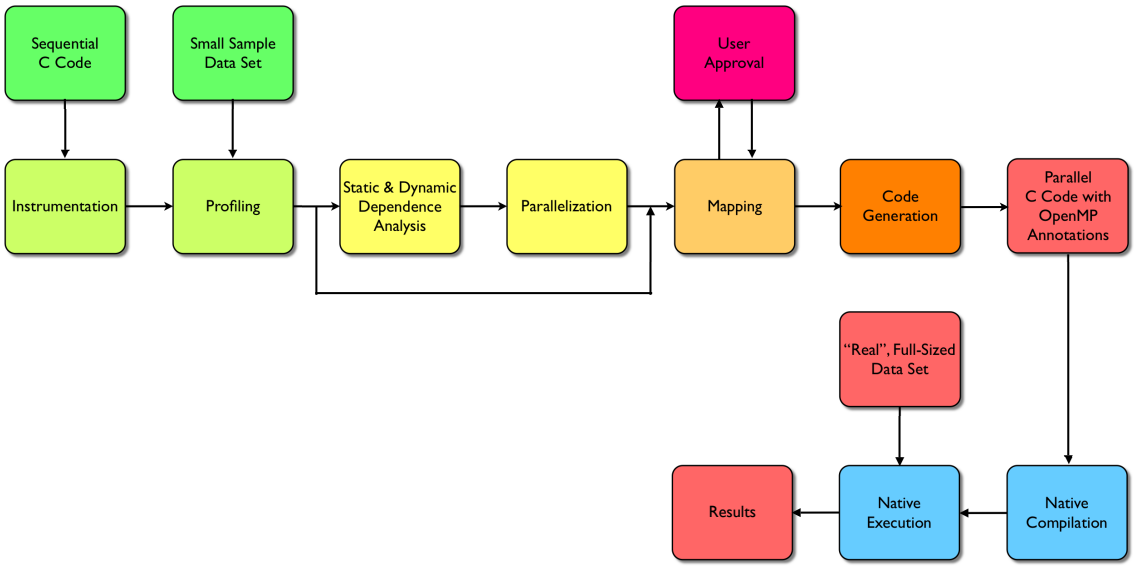


Figure 4. Our parallelization framework comprises IR-level instrumentation and profiling stages, followed by static and dynamic dependence analyses driving loop-level parallelization and a machine-learning based mapping stage where the user may be asked for final approval before parallel OpenMP code is generated. Platform-specific code generation is performed by the native OpenMP enabled C compiler.

has not "materialized" in the profiling run. In this case, we treat such a loop as potentially parallelizable, but present it to the user for final approval if parallelization is predicted to be profitable.

3.1.3 Parallel Code Generation

We use OpenMP for parallel code generation due to the low complexity of generating the required code annotations and the widespread availability of native OpenMP compilers. Currently, we only target parallel *FOR* loops and translate these into corresponding OpenMP annotations.

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 of 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. An analogous approach applies to privatizable arrays.

Reduction Operations. Reduction recognition for scalar variables is based on the algorithm presented in (22), but unlike the original publication we use a simplified code generation stage where it is sufficient to emit an OpenMP *reduction* annotation for each recognized reduction loop. We validate statically detected reduction candidates using profiling information and use an additional reduction template library to enable reductions on array locations such as that shown in figure 1.

Synchronization. The default behavior of parallel OpenMP loops is to synchronize threads at the end of the work sharing construct by means of a barrier. Due to the high cost of this form of synchronization it is important for good performance that redundant synchronization is avoided. Synchronization also increases idle time, due to load imbalance, and can sequentialize sections of a program. Based on the CDFG we compute inter-loop dependencies and apply a compile time barrier synchronization minimization algorithm (23), resulting in a minimal number of barriers. For those loops where the default synchronization can be eliminated we extend the annotations with the OpenMP *nowait* clause.

Limitations. At present, our approach to code generation is relatively simple and, essentially, relies on OpenMP code annotations alongside minor code transformations. We do not yet perform high-level code restructuring which might help expose or exploit more parallelism or improve data locality. While OpenMP is a compiler-friendly target for code generation it imposes a number of limitations. For example, we do not yet exploit coarse-grain parallelism, e.g. pipelines, and wavefront parallelism even though we can also extract this form of parallelism.

3.2 Machine Learning Based Parallelism Mapping

The responsibilities of the parallelism mapping stage are to decide if a parallel loop candidate is *profitable* to parallelize and, if so, to select a scheduling policy from the four options offered by OpenMP: *CYCLIC*, *DYNAMIC*, *GUIDED*, and *STATIC*. As the example in figure 2 demonstrates, this is a non-trivial task and the optimal solution depends on both the particular properties of the loop under consideration *and* the target platform. To provide a portable, but automated mapping approach we use a machine learning technique to construct a predictor that, after some initial training, will replace the highly platform-specific and often inflexible mapping heuristics of traditional parallelization frameworks.

3.2.1 Predictive Modeling

Separating profitably parallelizable loops from those that are not is a challenging task. Incorrect classification will result in missed opportunities for profitable parallel execution or even in a slowdown due to an excessive synchronization overhead. Traditional parallelizing compilers such as SUIF-1 employ simple heuristics based on the iteration count and the number of operations in the loop body to decide on whether or not a particular parallel loop candidate should be executed in parallel.

Our data – as shown in figure 5 – suggests that such a naïve scheme is likely to fail and that misclassification occurs frequently.

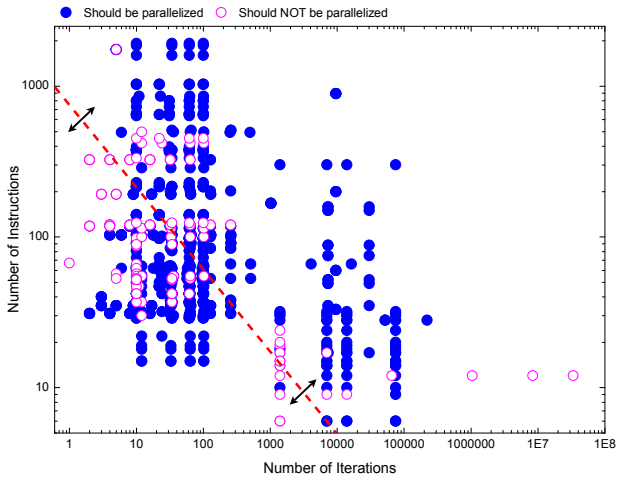


Figure 5. This diagrams shows the optimal classification (sequential/parallel execution) of all parallel loop candidates considered in our experiments for the Intel Xeon machine. Linear models and static features such as the iteration count and size of the loop body in terms of IR statements are not suitable for separating profitably parallelizable loops from those that are not.

A simple work based scheme would attempt to separate the profitably parallelizable loops by a diagonal line as indicated in the diagram in figure 5. Independent of where exactly the line is drawn there will always be loops misclassified and, hence, potential performance benefits wasted. What is needed is a scheme that (a) takes into account a richer set of – possibly dynamic – loop features, (b) is capable of non-linear classification, and (c) can be easily adapted to a new platform.

In this paper we propose a *predictive modeling* approach based on machine-learning classification. In particular, we use *Support Vector Machines (SVM)* (24) to decide (a) whether or not to parallelize a loop candidate and (b) how it should be scheduled. The SVM classifier is used to construct hyper-planes in the multi-dimensional space of *program features* – as discussed in the following paragraph – to identify profitably parallelizable loops. The classifier implements a multi-class SVM model with a *radial basis function (RBF)* kernel capable of handling both linear and non-linear classification problems (24). The details of our SVM classifier are provided in figure 6.

3.2.2 Program Features

We extract characteristic *program features* that sufficiently describe the relevant aspects of a program and present it to the SVM classifier. An overview of these features is given in table 1. The *static features* are derived from CoSy's internal code representation. Essentially, these features characterize the amount of work carried out in the parallel loop similar to e.g. (25). The *dynamic features* capture the dynamic data access and control flow patterns of the

Static features	IR Instruction Count IR Load/Store Count IR Branch Count Loop Iteration Count
Dynamic features	Data Access Count Instruction Count Branch Count

Table 1. Features characterizing each parallelizable loop.

1. Baseline SVM for classification
   (a) Training data:
   $\mathcal{D} = \{(\boldsymbol{x}_i, c_i)|\boldsymbol{x}_i \in \mathbb{R}^p, c_i \in \{-1, 1\}\}_{i=1}^n$
   (b) Maximum-margin hyperplane formulation:
   $c_i(\boldsymbol{w} \cdot \boldsymbol{x}_i - b) \geq 1$, for all $1 \leq i \leq n$.
   (c) Determine parameters by minimization of $||\boldsymbol{w}||$ (in $\boldsymbol{w}$, b) subject to 1.(b).
2. Extensions for non-linear multiclass classification
   (a) Non-linear classification:
   Replace dot product in 1.(b) by a kernel function, e.g. the following *radial basis function*:
   $k(\boldsymbol{x}, \boldsymbol{x}') = exp(-\gamma||\boldsymbol{x} - \boldsymbol{x}'||^2)$, for $\gamma > 0$.
   (b) Multiclass SVM:
   Reduce single multiclass problem into multiple binary problems. Each classifier distinguishes between one of the labels and the rest.

Figure 6. Support vector machines for non-linear classification.

sequential program and are obtained from the same profiling execution that has been used for parallelism detection.

3.2.3 Training Summary

We use an *off-line supervised learning* scheme whereby we present the machine learning component with pairs of program features and desired mapping decisions. These are generated from a library of known parallelizable loops through repeated, timed execution of the sequential and parallel code with the different available scheduling options and recording the actual performance on the target platform. Once the prediction model has been built using all the available training data, no further learning takes place.

3.2.4 Deployment

For a new, previously *unseen* application with parallel annotations the following steps need to be carried out:

1. *Feature extraction*. This involves collecting the features shown in table 1 from the *sequential* version of the program and is accomplished in the profiling stage already used for parallelism detection.
2. *Prediction*. For each parallel loop candidate the corresponding feature set is presented to the SVM predictor and it returns a classification indicating if parallel execution is profitable and which scheduling policy to choose. For a loop nest we start with the outermost loop ensuring that we settle for the most coarse-grained piece of work.
3. *User Interaction*. If parallelization *appears* to possible (according to the initial profiling) and profitable (according to the previous prediction step), but correctness cannot be proven by static analysis, we ask the user for his/her final approval.
4. *Code Generation*. In this step, we extend the existing OpenMP annotation with the appropriate scheduling clause, or delete the annotation if parallelization does not promise any performance improvement or has been rejected by the user.

3.3 Safety and Scalability Issues

Safety. Unlike static analysis, profile-guided parallelization cannot conclusively guarantee the absence of control and data dependences for *every possible* input. One simple approach regarding the selection of the "representative" inputs is based on control-flow

coverage analysis. This is driven by the empirical observation that for the vast majority of the cases the profile-driven approach might have a false positive ("there is a flow-dependence but the tool suggests the contrary") is due to a control-flow path that the data input set did not cover. This also gives a fast way to select representative workloads (in terms of data-dependencies) just by executing the applications natively and recording the resulting code coverage. Of course, there are many counter-examples where an input dependent data-dependence appears with no difference in the control-flow. The latter can be verified by the user.

For this current work, we have chosen a "worst-case scenario" and used the *smallest* data set associated with each benchmark for profiling, but evaluated against the *largest* of the available data sets. Surprisingly, we have found that this naive scheme has detected almost all parallelizable loops in the NAS and SPEC OMP benchmarks while not misclassifying any loop as parallelizable when it is not.

Furthermore, with the help of our tools we have been able to identify three incorrectly shared variables in the original NAS benchmarks that should in fact be privatized. This illustrates that manual parallelization is prone to errors and that automating this process contributes to program correctness.

Scalability. As we process data dependence information at byte-level granularity and effectively build a whole program CDFG 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 the largest applications encountered in our experimental evaluation. In comparison, static compilers that perform whole program analyses need to maintain similar data structures of about the same size. While the dynamic traces can potentially become very large as every single data access and control flow path is recorded, they can be processed *online*, thus eliminating the need for large traces to be stored.

As our approach operates at the IR level of the compiler we do not need to consider detailed architecture state, hence profiling can be accomplished at speeds close to native, sequential speed. For dependence analysis we only need to keep track of memory and control flow operations and make incremental updates to hash tables and graph structures. In fact, dependence analysis on dynamically constructed CDFGs has the same complexity as static analysis because we use the same representations and algorithms as the static counterparts.

4. Experimental Methodology

In this section we summarize our experimental methodology and provide details of the multi-core platforms and benchmarks used throughout the evaluation.

4.1 Platforms

We target both a shared memory (dual quad-core Intel Xeon) and distributed memory multi-core system (dual-socket QS20 Cell blade). A brief overview of both platforms is given in table 3.

4.2 Benchmarks

For our evaluation we have selected benchmarks (NAS and SPEC OMP) where both sequential and manually parallelized OpenMP versions are available. This has enabled us to directly compare our parallelization strategy against parallel implementations from independent expert programmers.

More specifically, we have used the NAS NPB (sequential v.2.3) and NPB (OpenMP v.2.3) codes (26) alongside the SPEC CPU2000 benchmarks and their corresponding SPEC OMP2001 counterparts. However, it should be noted that the sequential and parallel SPEC codes are not immediately comparable due to some amount of restructuring of the "official" parallel codes, resulting in a performance advantage of the SPEC OMP codes over the sequential ones, even on a single processor system.

Program	Suite	Data Sets/Xeon	Data Sets/Cell
BT	NPB2.3-OMP-C	S, W, A, B	NA
CG	NPB2.3-OMP-C	S, W, A, B	S, W, A
EP	NPB2.3-OMP-C	S, W, A, B	S, W, A
FT	NPB2.3-OMP-C	S, W, A, B	S, W, A
IS	NPB2.3-OMP-C	S, W, A, B	S, W, A
MG	NPB2.3-OMP-C	S, W, A, B	S, W, A
SP	NPB2.3-OMP-C	S, W, A, B	S, W, A
LU	NPB2.3-OMP-C	S, W, A, B	S, W, A
art	SPEC CFP2000	test, train, ref	test,train, ref
ammp	SPEC CFP2000	test, train, ref	test,train, ref
equake	SPEC CFP2000	test, train, ref	test,train, ref

Table 2. Benchmark applications and data sets.

Cell Blade Server	
Hardware	Dual Socket, QS20 Cell Blade 2×3.2 GHz IBM Cell processors 512KB L2 cache per chip 1GB XDRAM
O.S	Fedora Core 7 with Linux kernel 2.6.22 SMP
Compiler	IBM XLC single source compiler for Cell v0.9 -O5 -qstrict -qarch=cell -qipa=partition=minute (-qipa=overlay) Cell SDK 3.0
Intel Xeon Server	
Hardware	Dual Socket, Intel Xeon X5450 @ 3.00GHz 2 Quad-cores, 8 cores in total 6MB L2-cache shared/2 cores (12MB/chip) 16GB DDR2 SDRAM
O.S	64-bit Scientific Linux with kernel 2.6.9-55 x86_64
Compiler	Intel ICC 10.1 (Build 20070913) -O2 -xT -axT -ipo

Table 3. Hardware and software configuration details of the two evaluation platforms.

Each program has been executed using multiple different input data sets (shown in table 2), however, for parallelism detection and mapping we have only used the *smallest* of the available data sets[1]. The resulting parallel programs have then been evaluated against the larger inputs to investigate the impact of *worst-case input* on the safety of our parallelization scheme.

4.3 Methodology

We have evaluated three different parallelization approaches: *manual*, *auto-parallelization* using the Intel ICC compiler (just for the Intel platform), and our *profile-driven* approach.

For native code generation all programs (both sequential and parallel OpenMP) have been compiled using the Intel ICC and IBM XLC compilers for the Intel Xeon and IBM Cell platforms, respectively.

Furthermore, we use "leave-one-out cross-validation" to evaluate our machine-learning based mapping technique. This means that for K programs, we remove one, train a model on the remaining $K-1$ programs and predict the K^{th} program with the previously trained model. We repeat this procedure for each program in turn.

For the Cell platform we report parallel speedup over sequential code running on the general-purpose *PPE* rather than a single *SPE*. In all cases the sequential performance of the PPE exceeds that of

[1] Some of the larger data sets could not be evaluated on the Cell due to memory constraints.

a single SPE, ensuring we report improvements over the *strongest* baseline available.

5. Experimental Evaluation

In this section we present and discuss our results.

5.1 Overall Results

Figures 7(a) and 7(b) summarize our performance results for both the Intel Xeon and IBM Cell platforms.

Intel Xeon. The most striking result is that the Intel auto-parallelizing compiler fails to exploit any usable levels of parallelism across the whole range of benchmarks and data set sizes. In fact, auto-parallelization results in a slow-down of the *BT* and *LU* benchmarks for the smallest and for most data set sizes, respectively. ICC gains a modest speedup only for the larger data sets of the *IS* and *SP* benchmarks. The reason for this disappointing performance of the Intel ICC compiler is that it is typically parallelizing at inner-most loop level where significant fork/join overhead negates the potential benefit from parallelization.

The manually parallelized OpenMP programs achieve an average speedup of 3.5 across the benchmarks and data sizes. In the case of *EP*, a speedup of 8 was achieved for large data sizes. This is not surprising since this is an embarrassingly parallel program. More surprisingly, *LU* was able to achieve super-linear speedup ($9\times$) due to improved caching (27). Some programs (*BT*, *MG* and *CG*) exhibit lower speedups with larger data sets (A and B in comparison to W) on the Intel machine. This is a well-known and documented scalability issue of these specific benchmarks (28; 27).

For most NAS benchmarks our profile-driven parallelization achieves performance levels close to those of the manually parallelized versions, and sometimes outperforms them (*EP*, *IS* and *MG*). This surprising performance gain can be attributed to three important factors. Firstly, our approach parallelizes outer loops whereas the manually parallelized codes have parallel inner loops. Secondly, our approach exploits reduction operations on array locations and, finally, the machine learning based mapping is more accurate in eliminating non-profitable loops from parallelization and selecting the best scheduling policy.

The situation is slightly different for the SPEC benchmarks. While profile-driven parallelization still outperforms the static auto-parallelizer we do not reach the performance level of the manually parallelized codes. Investigations into the causes of this behavior have revealed that the SPEC OMP codes are not equivalent to the sequential SPEC programs, but have been manually restructured (29). For example, data structures have been altered (e.g. from *list* to *vector*) and standard memory allocation (excessive use of *malloc*) has been replaced with a more efficient scheme. Obviously, these changes are beyond what an auto-parallelizer is capable of performing. In fact, we were able to confirm that the sequential performance of the SPEC OpenMP codes is on average about 2 times (and up to 3.34 for *art*) above that of their original SPEC counterparts. We have verified that our approach parallelizes the same critical loops for both *equake* and *art* as SPEC OMP. For *art* we achieve a speedup of 4, whereas the SPEC OMP version is 6 times faster than the sequential SPEC FP version, of which more than 50% is due to sequential code optimizations. We also measured the performance of the profile-driven parallelized *equake* version using the same code modifications and achieved a comparable speedup of 5.95.

Overall, the results demonstrate that our profile-driven parallelization scheme significantly improves on the state-of-the-art Intel auto-parallelizing compiler. In fact, our approach delivers performance levels close to or exceeding those of manually parallelized codes and, on average, we achieve 96% of the performance of hand-tuned parallel OpenMP codes, resulting in an average speedup of 3.34 across all benchmarks.

IBM Cell. Figure 7(b) shows the performance resulting from manual and profile-driven parallelization for the dual-Cell platform.

Unlike the Intel platform, the Cell platform does not deliver a high performance on the manually parallelized OpenMP programs. On average, these codes result in an overall slowdown. For some programs such as *CG* and *EP* small performance gains could be observed, however, for most other programs the performance degradation is disappointing. Given that these are hand-parallelized programs this is perhaps surprising and there are essentially two reasons why the Cell's performance potential could not be exploited. Firstly, it is clear that the OpenMP codes have not been developed specifically for the Cell. The programmer have not considered the communication costs for a distributed memory machine. Secondly, in absence of specific scheduling directives the OpenMP runtime library resorts to its default behavior, which leads to poor overall performance. Given that the manually parallelized programs deliver high performance levels on the Xeon platform, the results for the Cell demonstrate that parallelism detection in isolation is not sufficient, but mapping must be regarded as equally important.

In contrast to the "default" manual parallelization scheme, our integrated parallelization strategy is able to successfully exploit significant levels of parallelism, resulting in average speedup of 2.0 over the sequential code and up to 6.2 for individual programs (*EP*). This success can largely be attributed to the improved mapping of parallelism resulting from our machine-learning based approach.

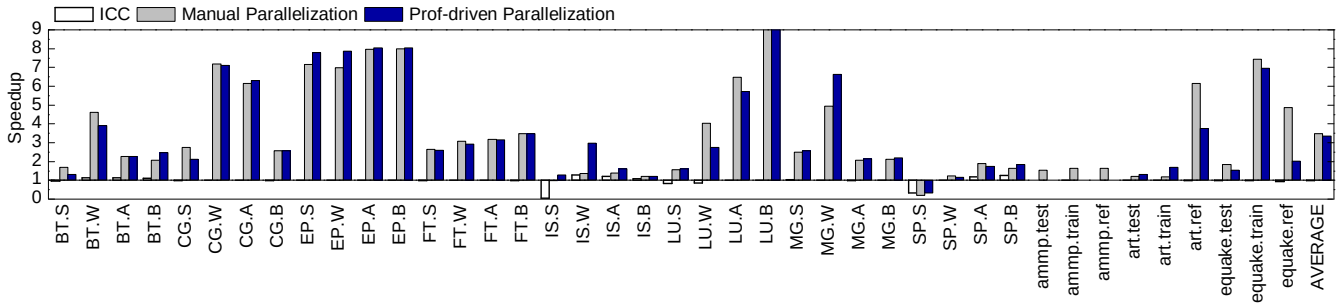
5.2 Parallelism Detection and Safety

Our approach relies on dynamic profiling information to discover parallelism. This has the obvious drawback that it may classify a loop as potentially parallel when there exists another data set which would highlight a dependence preventing correct parallelization. This is a fundamental limit of dynamic analysis and the reason for requesting the user to confirm uncertain parallelization decisions. It is worthwhile, therefore, to examine to what extent our approach suffers from *false positives* ("loop is incorrectly classified as parallelizable"). Clearly, an approach that suffers from high numbers of such false positives will be of limited use to programmers.

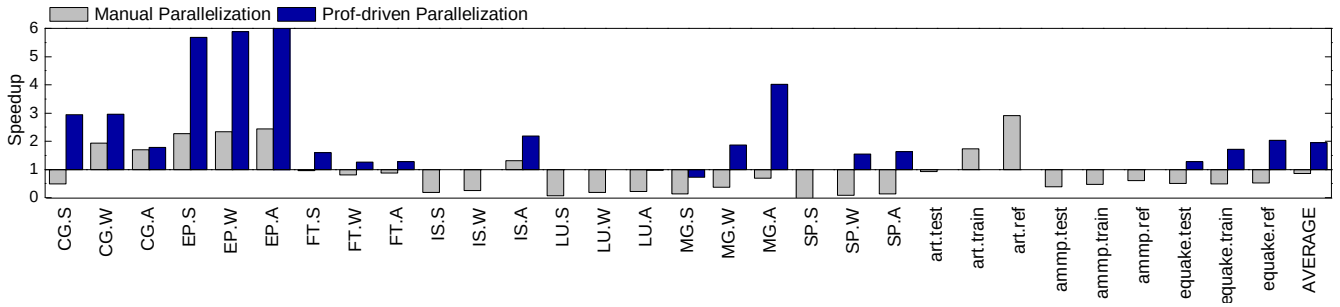
Column 2 in table 5.2 shows the number of loops our approach detects as potentially parallel. The column labeled *FP* ("false positive") shows how many of these were in fact sequential. The surprising result is that none of the loops we considered potentially parallel turned out to be genuinely sequential. Certainly, this results does not prove that dynamic analysis is always correct. Still, it indicates that profile-based dependence analysis may be more accurate than generally considered, even for profiles generated from small data sets. Clearly, this encouraging result will need further validation on more complex programs before we can draw any final conclusions.

Column 3 in table 5.2 lists the number of loops parallelizable by ICC. In some applications, the ICC compiler is able to detect a considerable number of parallel loops. In addition, if we examine the coverage (shown in parentheses) we see that in many cases this covers a considerable part of the program. Therefore we conclude that it is less a matter of the parallelism detection that causes ICC to perform so poorly, but rather how it exploits and maps the detected parallelism (see section 5.3).

The final column in table 5.2 eventually shows the number of loops parallelized in the hand-coded applications. As before, the percentage of sequential coverage is shown in parentheses. Far fewer loops than theoretically possible are actually parallelized because the programmer have obviously decided only to parallelize those loops they considered "hot" and "profitable". These loops



(a) Speedup over sequential codes achieved by ICC auto-parallelization, manual parallelization and profile-driven parallelization for the Xeon platform.



(b) Speedup over sequential code achieved by manual parallelization and profile-driven parallelization for the dual Cell platform.

Figure 7. Speedups due to different parallelization schemes.

	Profile driven			ICC no threshold	Manual
Application	#loops(%cov)	FP	FN	#loops(%cov)	#loops(%cov)
bt	205 (99.9%)	0	0	72 (18.6%)	54 (99.9%)
cg	28 (93.1%)	0	0	16 (1.1%)	22 (93.1%)
ep	8 (99.9%)	0	0	6 (<1%)	1 (99.9%)
ft	37 (88.2%)	0	0	3 (<1%)	6 (88.2%)
is	9 (28.5%)	0	0	8 (29.4%)	1 (27.3%)
lu	154 (99.7%)	0	0	88 (65.9%)	29 (81.5%)
mg	48 (77.7%)	0	3	9 (4.7%)	12 (77.7%)
sp	287 (99.6%)	0	0	178 (88.0%)	70 (61.8%)
equake_SEQ	69 (98.1%)	0	0	29 (23.8%)	11 (98.0%)
art_SEQ	31 (85.6%)	0	0	16 (30.0%)	5 (65.0%)
ammp_SEQ	21 (1.4%)	0	1	43 (<1%)	7 (84.4%)

Table 4. Number of parallelized loops and their respective coverage of the sequential execution time.

cover a significant part of the sequential time and effective parallelization leads to good performance as can be seen for the Xeon platform.

In total there are four *false negatives* (column *FN* in table 5.2) , i.e. loops not identified as parallel although safely parallelizable. Three false negatives are contained in the *MG* benchmark, and two of these are due to loops which have zero iteration counts for all data sets and, therefore, are never profiled. The third one is a *MAX* reduction, which is contained inside a loop that our machine-learning classifier has decided not to parallelize.

5.3 Parallelism Mapping

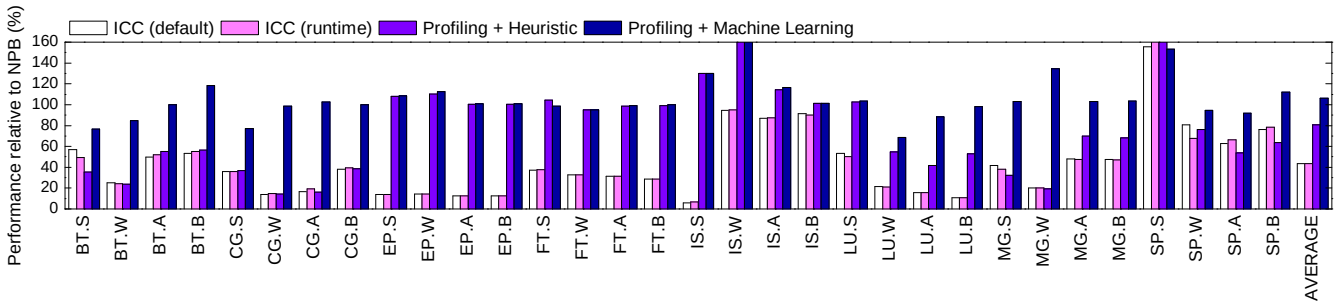
In this section we examine the effectiveness of three mapping schemes (manual, heuristic with static features, and machine-learning using profiling information) across the two platforms.

Intel Xeon. Figure 8(a) compares the performance of ICC and our approach to that of the hand-parallelized OpenMP programs. In the case of ICC we show the performance of two different mapping approaches. By default, ICC employs a compile-time profitability check while the second approach performs a runtime check using a dynamic profitability threshold.

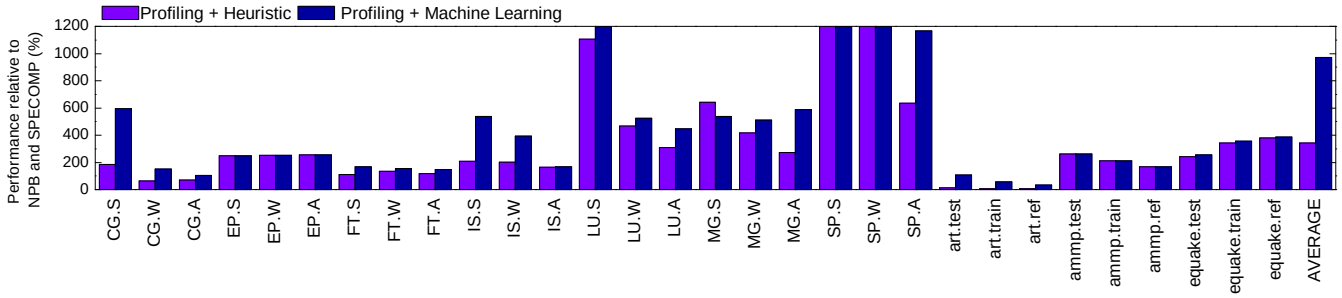
For some cases (*BT.B* and *SP.B*) the runtime checks provide a marginal improvement over the static mapping scheme while the static scheme is better for *IS.B*. Overall, both schemes are equally poor and deliver less than half of the speedup levels of the hand-parallelized benchmarks. The disappointing performance appears to be largely due to non-optimal mapping decisions, i.e. to parallelize inner loops rather than outer ones.

In the same figure we compare our machine-learning based mapping approach against a scheme which uses the same profiling information, but employs a fixed, work-based *heuristic* similar to the one implemented in the SUIF-1 parallelizing compiler (see also figure 5). This heuristic considers the product of the iteration count and the number of instructions contained in the loop body and decides against a static threshold. While our machine-learning approach delivers nearly the performance of the hand-parallelized codes and, in some cases, is able to outperform them, the static heuristic performs poorly and is unable to obtain more than 85% of the performance of the hand-parallelized code. This translates into an average speedup of 2.5 rather than 3.7 for the NAS benchmarks. The main reason for this performance loss is that the default scheme using only static code features and a linear work model is unable to accurately determine whether a loop should be parallelized or not.

In figure 9 we compare the performance resulting from the different automated mapping approaches to that of the hand-parallelized SPEC OMP codes. Again, our machine-learning based approach outperforms ICC and the fixed heuristic. On average, our



(a) NAS benchmarks on the Intel Xeon platform.



(b) NAS and SPEC FP benchmarks on the IBM Cell platform.

Figure 8. Impact of different mapping approaches (100% = manually parallelized OpenMP code).

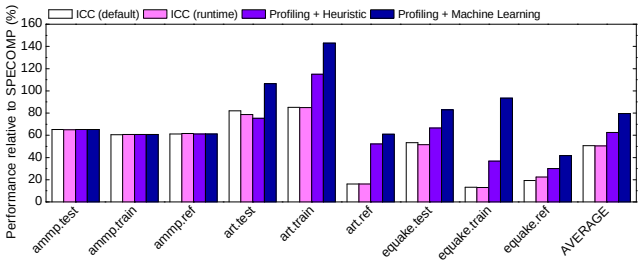


Figure 9. Impact of different mapping approaches for the SPEC benchmarks (100% = manually parallelized OpenMP code).

approach delivers 88% of the performance of the hand-parallelized code, while ICC and the fixed heuristic approach achieve performance levels of 45% and 65%, respectively. The lower performance gains for the SPEC benchmarks are mainly due to a better starting point of the hand-parallelized SPEC OMP benchmarks (see section 5.1).

IBM Cell. The diagram in figure 8(b) shows the speedup of our machine-learning based mapping approach over the hand-parallelized code on the Cell platform. As before, we compare our approach against a scheme which uses the profiling information, but employs a fixed mapping heuristic.

The manually parallelized OpenMP programs are not specifically "tuned" for the Cell platform and perform poorly. As a consequence, the profile-based mapping approaches show high performance gains over this baseline, in particular, for the small input data sets. Still, the combination of profiling and machine-learning outperforms the fixed heuristic counterpart by far and, on average, results in a speedup of 9.7 over the hand-parallelized OpenMP programs across all data sets.

Summary The combined profiling and machine-learning approach to mapping comes within reach of the performance of hand-parallelized code on the Intel Xeon platform and in some cases outperforms it. Fixed heuristics are not strong enough to separate profitably parallelizable loops from those that are no and perform poorly. Typically, static mapping heuristics result in performance levels of less than 60% of the machine learning approach. This is because the default scheme is unable to accurately determine whether a loop should be parallelized or not. The situation is exacerbated on the Intel Cell platform where accurate mapping decisions are key enablers to high performance. Existing ("generic") manually parallelized OpenMP codes fail to deliver any reasonable performance and heuristics, even if based on profiling data, are unable to match the performance of our machine-learning based scheme.

5.4 Scalability

For the Xeon platform the *LU* and *EP* benchmarks scale well with the number of processors (see figure 10). In fact, a super-linear speedup due to more cache memory in total can be observed for the *LU* application. For other benchmarks scalability is more limited and often saturation effects occur for four or more processors. This scalability issue of the NAS benchmarks is well-known and in line with other research publications (27). Figure 11 shows a performance drop for the step from one to two processors on the Cell platform. This is due to the fact that we use the generally more powerful PPE to measure single processor performance, but then use the multiple SPEs for parallel performance measurements. The diagram reveals that in the best case it takes about three SPEs to achieve the original performance of the PPE. Some of the more scalable benchmarks such as *EP* and *MG* follow a linear trend as

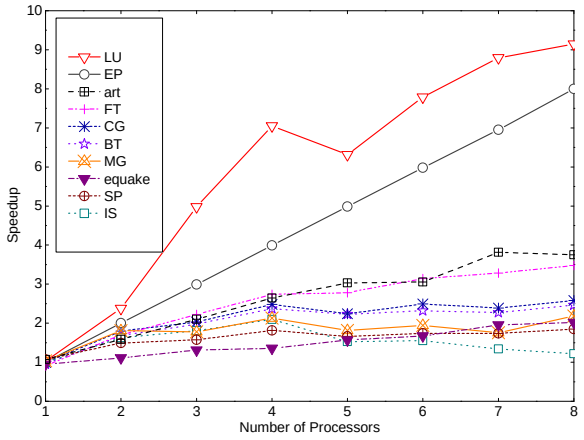


Figure 10. Scalability on the Intel platform (largest data set).

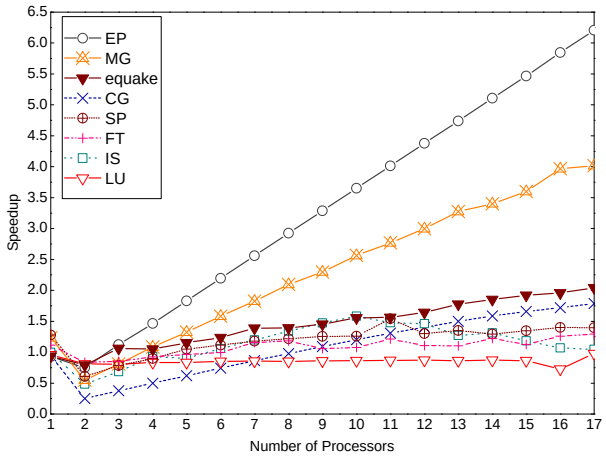


Figure 11. Scalability on the Cell platform (largest data set).

the number of processors increases, however, most of the remaining benchmarks saturate at a low level.

6. Related Work

Parallel Programming Languages. Many approaches have been proposed for changing or extending to existing programming languages to enable easier exploiting of parallelism (9; 10; 12). Unfortunately, these approaches do not alleviate all problems of porting legacy sequential programs.

Automatic Parallelization. Static automatic parallelism extraction has been achieved on restricted DOALL and DOACROSS loops (3; 4; 5). Unfortunately, many parallelization opportunities could still not be discovered by a static analysis approach due to lack of information at the source code level.

Speculative Parallelization. There are existing automatic parallelization techniques that exploit parallelism in a speculatively execution manner (30; 31; 32), but these approaches typically require hardware support. Matthew *et al.* (33) have manually parallelized the SPECINT-2000 benchmarks with thread level speculation. Their approach relies upon the programmer to discover parallelism as well as runtime support for parallel execution.

Dynamic Parallelization Rus et al. (34) applied sensitivity analysis to automatic parallelize programs whose behaviors may be sensitive to input data sets. In contrast to their static analysis and runtime checking approach, our profiling-driven approach discovers more parallel opportunities as well as selecting parallel candidates and scheduling policies. Dynamic dependence analysis (35; 36) and hybrid data dependence analysis (37) make use of dynamic dependence information, but delay much of the parallelization work to the runtime of the program. In contrast, we employ a separate profiling stage and incorporate the dynamic information in the usual compiler based parallelization without causing any runtime overhead.

Interactive Parallelization. Interactive parallelization tools (13; 17; 18; 19; 20) provide a way to actively involve the programmer in the detection and mapping of application parallelism. For example, SUIF Explorer (13) helps the programmer to identify those loops that are likely to be parallelizable and assists the user in checking for correctness. Similarly, the *Software Behavior-Oriented Parallelization* (38) system allows the programmer to specify intended parallelism. In (39), programmers mark potential parallel regions of the program, and the tool uses dynamic profiling information to find a good mapping of parallel candidates. Unlike our approach, these frameworks require the programmer to mark parallel regions, instead of discovering parallelism automatically. Moreover, the problem of mapping parallelism across architectures is not well addressed in these approaches.

Parallelism Mapping. Prior research in parallelism mapping has mainly focused on building heuristics and analytical models (40; 41), runtime adaptation (42; 43) approaches, and on mapping or migrating tasks on a specific platform. Instead of proposing a new scheduling or mapping technique for a particular platform, we aim to develop a compiler-based, automatic, and portable approach that learns how to take advantage of existing compilers and runtime system for efficiently mapping parallelism. Ramanujam and Sadayappan (40) used heuristics to solve the task mapping problems in distributed memory machines. Their model requires low-level detail of the hardware platform, such as the communication cost, which have to be re-tuned once the underlying architecture changes. Static analytical models have been proposed for predicting the program's behaviors. For example, the OpenUH compiler uses a cost model for evaluating the cost for parallelizing OpenMP programs (41). There are also some models that predict the parallel performance such as LogP (44). However, these models require help from their users and are not portable. Corbalan *et al.* (42) measure performance and allocate processors during runtime. The adaptive loop scheduler (43) selects both the number of threads and the scheduling policy for a parallel region in SMPs through runtime decisions. In contrast to this runtime approach, this paper presents a static processor allocation scheme which is performed at compilation time.

Adaptive Compilation. Machine learning and statistical methods have already been used in single core program transformation. For example, Cooper *et al.* (45) develop a technique to find "good" compiler optimization sequences for code size reduction.

In contrast to prior research, we built a model that learns how to effectively map parallelism to multi-core platforms with existing compilers and runtime systems. The model is automatically constructed and trained off-line, and the parallelism decisions are made at the compilation time.

7. Conclusion and Future Work

In this paper we have developed a platform-agnostic, profiling-based parallelism detection method that enhances static data dependence analyses with dynamic information, resulting in larger

amounts of parallelism uncovered from sequential applications. We have also shown that parallelism detection in isolation is not sufficient to achieve high performance, but requires close interaction with an adaptive mapping scheme to unfold the full potential of parallel execution across programs and architectures.

Results obtained on two complex multi-core platforms (Intel Xeon and IBM Cell) and two sets of benchmarks (NAS and SPEC) confirm that our method is more aggressive in parallelization and more portable than existing static auto-parallelization and achieves performance levels close to manually parallelized codes.

Future work will focus on further improvements of the profiling-based data dependence analysis with the ultimate goal of eliminating the need for the user's approval for parallelization decisions that cannot be proven conclusively. Furthermore, we will integrate support for restructuring transformations into our framework and target parallelism beyond the loop level.

References

[1] H. P. Hofstee. Future microprocessors and off-chip SOP interconnect. *IEEE Trans. on Advanced Packaging*, 27(2), May 2004.

[2] L. Lamport. The parallel execution of DO loops. *Communications of ACM*, 17(2), 1974.

[3] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *PLDI* , 1986.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[5] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. *Parallel Computing*, ACM, 1997.

[6] D. A. Padua, R. Eigenmann, et al. Polaris: A new-generation parallelizing compiler for MPPs. Technical report, In CSRD No. 1306. UIUC, 1993.

[7] M. W. Hall, J. M. Anderson, et al. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12), 1996.

[8] Open64. `http://www.open64.net`.

[9] F. Matteo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. *PLDI*, 1998.

[10] M. Gordon, W. Thies, M. Karczmarek, et al. A stream compiler for communication-exposed architectures. *ASPLOS*, 2002.

[11] P. Husbands Parry, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. *SC*, 2003.

[12] V. A. Saraswat, V. Sarkar, and C von. Praun. X10: Concurrent programming for modern architectures. *PPoPP*, 2007.

[13] L. Shih-Wei, D. Amer, et al. SUIF Explorer: An interactive and interprocedural parallelizer. *SIGPLAN Not.*, 34(8), 1999.

[14] M. Kulkarni, K. Pingali, B. Walter, et al. Optimistic parallelism requires abstractions. *PLDI'07*, 2007.

[15] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. *Inter. Workshop LCR*, 1998.

[16] Jia Guo, Ganesh Bikshandi, et al. Hierarchically tiled arrays for parallelism and locality. *IPDPS*, 2006.

[17] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. *ICS* 1991

[18] K. Kennedy, K. S. McKinley, and C. W. Tseng. Interactive parallel programming using the Parascope editor. *IEEE TPDS*, 2(3), 1991.

[19] T. Brandes, S. Chaumette, M. C. Counilh et al. HPFIT: a set of integrated tools for the parallelization of applications using high performance Fortran. part I: HPFIT and the Transtool environment. *Parallel Comput.*, 23(1-2), 1997.

[20] M. Ishihara, H. Honda, and M. Sato. Development and implementation of an interactive parallelization assistance tool for OpenMP: iPat/OMP. *IEICE Trans. Inf. Syst.*, E89-D(2), 2006.

[21] S. Rul, H. Vandierendonck, and K. De Bosschere. A dynamic analysis tool for finding coarse-grain parallelism. In *HiPEAC Industrial Workshop*, 2008.

[22] W. M. Pottenger. Induction variable substitution and reduction recognition in the Polaris parallelizing compiler. Technical Report, UIUC, 1994.

[23] M. O'Boyle and E. Stöhr. Compile time barrier synchronization minimization. *IEEE TPDS*, 13(6), 2002.

[24] E. B. Bernhard, M. G. Isabelle, and N. V. Vladimir. A training algorithm for optimal margin classifiers. *Workshop on Computational Learning Theory*, 1992.

[25] H. Ziegler and M. Hall. Evaluating heuristics in automatically mapping multi-loop applications to FPGAs. *FPGA*, 2005.

[26] D. H. Bailey, E. Barszcz, et al. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3), 1991.

[27] R. E. Grant and A. Afsahi. A Comprehensive Analysis of OpenMP Applications on Dual-Core Intel Xeon SMPs. *IPDPS*, 2007.

[28] *NAS Parallel Benchmarks 2.3, OpenMP C version*. http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html.

[29] V. Aslot, M. Domeika, et al. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. *LNCS*, 2001.

[30] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. *ISCA*, 1998.

[31] J. Dou and M. Cintra. Compiler estimation of load imbalance overhead in speculative parallelization. *PACT*, 2004.

[32] R. Ramaseshan and F. Mueller. Toward thread-level speculation for coarse-grained parallelism of regular access patterns. *MULTIPROG*, 2008.

[33] M. Bridges, N. Vachharajani, et al. Revisiting the sequential programming model for multi-core. *MICRO*, 2007.

[34] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores, 2007. *ICS*, 2007

[35] P. Peterson and D. Padua. Dynamic dependence analysis: A novel method for data dependence evaluation. *LCPC*, 1992.

[36] M. Chen and K. Olukotun. The JRPM system for dynamically parallelizing Java programs. *ISCA*, 2003.

[37] S. Rus and L. Rauchwerger. Hybrid dependence analysis for automatic parallelization. Technical Report, Dept. of CS, Texas A&M U., 2005.

[38] C. Ding, X. Shen, et al. Software behavior oriented parallelization. *PLDI*, 2007.

[39] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. *MICRO*, 2007.

[40] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. *SC*, 1989.

[41] C. Liao and B. Chapman. A compile-time cost model for OpenMP. *IPDPS*, 2007.

[42] J. Corbalan, X. Martorell, and J. Labarta. Performance-driven processor allocation. *IEEE TPDS*, 16(7), 2005.

[43] Y. Zhang and M. Voss. Runtime empirical selection of loop schedulers on Hyperthreaded SMPs. *IPDPS*, 2005.

[44] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[45] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *LCTES*, 1999.

[46] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. *Artificial Intelligence: Methodology, Systems, Applications*, 2002.

[47] L.N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: part II, multidimensional time. *PLDI*, 2008.