A

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

ZHENG WANG, Lancaster University, United Kingdom
GEORGIOS TOURNAVITIS, Intel Barcelona Research Center, Spain
BJÖRN FRANKE and MICHAEL F. P. O'BOYLE, University of Edinburgh, United Kingdom

Compiler-based auto-parallelization is a much studied area, but has yet to find wide-spread application. This is largely due to the poor identification and exploitation of application parallelism, resulting in disappointing performance far below that 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 the identification of more application parallelism and only rely on the user for final approval. We then replace the traditional target-specific and inflexible mapping heuristics with a machine-learning based prediction mechanism, resulting in better mapping decisions while automating adaptation to different target architectures. We have evaluated our parallelization strategy on the NAS and SPEC CPU2000 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, Optimization*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms: Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: Auto-Parallelization, Profile-Driven Parallelism Detection, Machine-Learning Based Parallelism Mapping, OpenMP

ACM Reference Format:
Z. Wang et al. 2013. Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping ACM Trans. Architec. Code Optim. V, N, Article A (January YYYY), 25 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Multi-core computing systems are widely seen as the most viable means of delivering performance with increasing transistor densities [Asanovic et al. 2009]. However, this potential cannot be realized unless the application has been well parallelized. Unfortunately, efficient parallelization of a sequential program is a challenging and

This article extends [Tournavitis et al. 2009] published in PLDI '09.
Work conducted in part while Z. Wang and G. Tournavitis were with the University of Edinburgh.
Authors' addresses: Z. Wang (corresponding author), School of Computing and Communications, Lancaster University, UK; email: z.wang@lancaster.ac.uk; G. Tournavitis, Intel Barcelona Research Center, Intel Labs Barcelona, Spain; B. Franke and M. F. P. O'Boyle, School of Informatics, the University of Edinburgh, UK.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

error-prone task. It is widely acknowledged that manual parallelization by expert programmers results in the most efficient parallel implementation, but is a costly and time-consuming approach. Parallelizing compiler technology, on the other hand, has the potential to greatly reduce this cost. As hardware parallelism increases in scale with each generation and programming costs increase, parallelizing technology becomes extremely attractive. However, despite the intense research interest in the area, it has failed to deliver outside niche domains.

Automatic parallelism extraction is certainly not a new research area [Lamport 1974]. Progress was achieved in 1980s to 1990s on restricted *DOALL* and *DOACROSS* loops [Burke and Cytron 1986; Lim and Lam 1997; Kennedy and Allen 2002]. In fact, this research has resulted in a whole range of parallelizing research compilers, e.g. Polaris [Padua et al. 1993], SUIF-1 [Hall et al. 1996] and, more recently, Open64 [Open64 2013]. Complementary to the on-going work in auto-parallelization many high-level parallel programming languages – such as Cilk [Frigo et al. 1998], OpenMP, UPC [Husbands et al. 2003] and X10 [Saraswat et al. 2007] – and programming models – such as STAPL [Rauchwerger et al. 1998] and Galois [Kulkarni et al. 2007] – have been proposed. Interactive parallelization tools [Irigoin et al. 1991; Kennedy et al. 1991; Brandes et al. 1997; Ishihara et al. 2006] 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 [Gordon 2010], the effort involved in discovering and mapping parallelism is still far greater than that of writing an equivalent sequential program.

This article 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 dependence and enhance the corresponding static analysis with dynamic information. Subsequently, we apply an off-line trained machine-learning based prediction mechanism to each parallel loop candidate and decide if and how the parallel mapping should be performed[1]. 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 automatically by the compiler.

Results. We have evaluated our parallelization strategy against the NAS and SPEC CPU2000 benchmarks and two different multi-core platforms (Intel Xeon SMP and IBM 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 analysis 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

[1] In this article we are interested in determining whether it is profitable to parallelize a loop and how the loop should be scheduled if a parallel execution is profitable.

```
for (i = 0; i < nodes; i++) {
  Anext = Aindex[i];     //The value of Anext is unknown at compile time.
  Alast = Aindex[i + 1];

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

  while (Anext < Alast) {
    col = Acol[Anext];  //The value of col is unknown at compile time.
                        //It is used for indexing both arrays v and w.
    sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] + ...;
    sum1 += ...
    w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + ...;
    w[col][1] += ...

    Anext++;
  }

  w[i][0] += sum0;
  ...
}
```

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

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-parallelized 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 article 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 sections 3, 4 and 5. Section 6 provides further discussions about the safety and scalability issues of our framework. Our experimental methodology and results are discussed in sections 7 and 8, respectively. We establish a wider context of related work in section 9 before we summarize and conclude in section 10.

2. MOTIVATION

In this section, we first show that static analysis can be overly conservative in detecting parallelism; then, we demonstrate that the parallelism mapping decision has a significant impact on performance and the right mapping may vary across different architectures.

2.1. 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. Conservative, static analysis fails to parallelize both loops due to sparse matrix operations with indirect array indices and the inner *while* loop. In fact, w is passed as a pointer argument and it accesses memory allocated at the same pro-

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

Fig. 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.

gram point with v. In addition, the indirect array access (a common programming technique), e.g. the index of w is col (line 21 in figure 1) that is determined by $Acol[Anext]$ (line 13 of figure 1), makes most of the static dependence tests inconclusive for determining data dependence.

Profiling-based dependence analysis, on the other hand, provides us with the additional information that no actual data dependence inhibits parallelization for a given sample input. This is useful for auto-parallelizing programs with data-independent parallelism which is difficult to discover using static compiler analysis. By instrumenting every dynamic memory access, one can discover that all the read and write statements of array w are in fact commutative and associative for the specific input. Hence the iterations of the inner and outer loops can be executed in parallel given that the partial sums values will be accumulated to w after the execution of the loop. While we still cannot prove absence of data dependence 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. 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 potential application parallelism than purely static approaches.

2.2. Parallelism 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 Broadband Engine (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. OVERVIEW OF THE FRAMEWORK

This section provides an overview of our parallelization framework, which combines profiling-driven parallelism detection and a machine-learning based mapping model to generate optimized parallel code.

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 addition, data scoping for shared and private data also

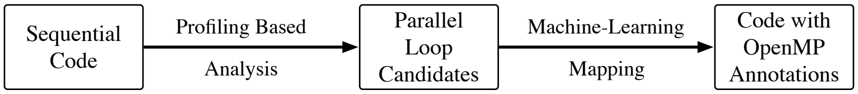


Fig. 3. A two-staged parallelization approach.

takes place at this stage. Our implementation targets data parallel loops. Other types of parallelism (e.g. task parallelism) are not currently supported.

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. Parallelism annotations are also removed for loop candidates where correctness cannot be proven conclusively (based on static analysis) and the user disapproves of the suggested parallelization decision. Our model currently evaluates each candidate loop in isolation. Finally, the parallel code is compiled with a native OpenMP compiler for the target platform. A complete overview of our tool-chain is shown in figure 4.

In the following two sections, we describe our novel framework in detail. We first introduce the use of profiling information to detect parallelism and generate code. This is followed by a description of the machine-learning based mapping model.

4. PARALLELISM DETECTION AND CODE GENERATION.

We propose a profile-driven approach for parallelism detection where the traditional static compiler analysis is not replaced, but *enhanced* with dynamic information. To achieve this we have devised an instrumentation scheme *operating at the intermediate representation (IR) level* of the compiler. Unlike e.g. [Rul et al. 2008], 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 four stages involved in parallelism detection are:

(1) IR instrumentation and profile generation
(2) `CDFG` construction
(3) Parallelism detection
(4) Parallel Code generation

We describe the profiling-driven approach by following these four stages in turn.

4.1. 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 [CoSy 2009]). 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 insert 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 compiler. Because the

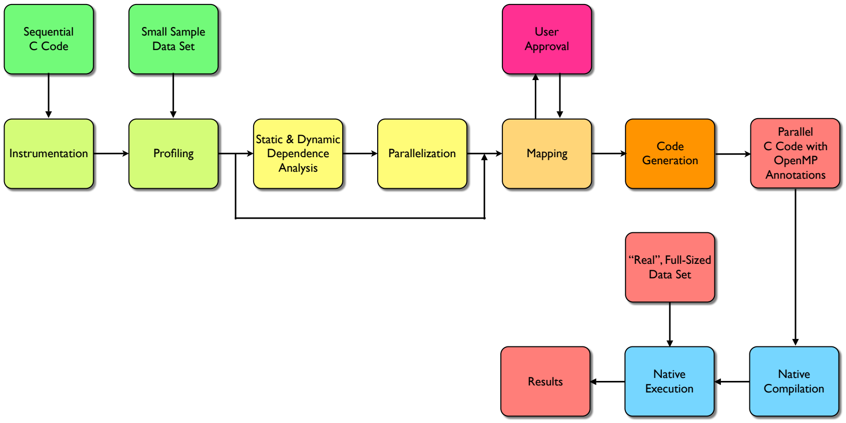


Fig. 4. Our parallelization framework comprises IR-level instrumentation and profiling stages, followed by static and dynamic dependence analyses driving loop-level parallelization and a machine-learning based mapping stage where the user may be asked for final approval before parallel OpenMP code is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

available parallelism is often only dependent on the program itself, the profiling and analysis can be done on any general purposed platforms. In this work, the sequential program was profiled on each target platform.

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

4.2. `CDFG` Construction

The instrumented program is now run and the profile data generated. Using this profiling information, our framework automatically constructs a global control and data flow graph (`CDFG`) on which parallelism detection is performed. Since the subsequent dependence analysis stage consumes one item of the profiling trace at a time, the `CDFG` can be constructed incrementally. Hence, it is not necessary to store the entire trace if the tools are chained up appropriately.

The profiling information, i.e. trace, is processed by algorithm 1 that performs dependence analysis. This algorithm distinguishes between control and data flow items and maintains various data structures supporting dependence analysis. The control flow section (lines $4-13$) constructs a global control flow graph of the application including call stacks, loop nest trees, and normalized loop iteration vectors. The data-flow section (lines $14-25$) is responsible for mapping memory addresses to specific high-level data flow information. For this we keep a hash table (lines $16-17$) where data items are traced at byte-level granularity. Data dependences are recorded as data edges in the `CDFG` (lines $18-25$). These edges are further annotated with the specific data sections (e.g. array indices) that cause the dependence. For loop-carried data dependency an additional bit vector relating the dependence to the surrounding loop nest is maintained.

As soon as the complete trace has been processed, the constructed `CDFG` with all its associated annotations is imported back into the 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 incomplete dependence information, for example, if a *may* data dependence has not materialized in the profiling run. In this case, we treat

Algorithm 1: Algorithm for CDFG construction.

```
Data
· CDFG($V, E_C, E_D$): graph with control ($E_C$) and data-flow ($E_D$) edges
· $loop\_carried_e$[]: bitset $\forall e \in E_D$: $loop\_carried_e[i] = 1$ if $e$ loop-carried in loop-level $i$
· $set_e$[]: address-range array $\forall e \in E_D$, indexed by the loop-carried level $i$
· $it_a$[]: iteration vector of address $a$
· $M[A, \{V, it_a\}]$: hash table: memory addr. $a \rightarrow \{V, it_a\}$
· $I(k)$: extract field $k$ from instruction $I$
· $GD$: global memory address-range tree
· $D_f$: memory address-range tree of function $f$
· $it_0$[]: current normalised iteration vector
· $u \in V$: current node
· $f \in V$: current function
· $l \in V$: current loop
· $c \in V$: current component

Procedure $IR\_instruction\_handler$
while trace not finished do
    $I \leftarrow$ next instruction;
    if $I$ is a control instruction then
        if I(id) $\notin c$ then
            create node $v$ for I(id) in $c$;
        if edge $(u, v) \notin E_C$ then
            add $(u, v)$ in CDFG ;
        switch $I$ do
            case bb  $u \leftarrow v$;
            case func  $f \leftarrow v$;
            case loop  $l \leftarrow v$;
            case iter  $it_0[depth(l)] \leftarrow it_0[depth(l)] + 1$;
    else if $I$ is a memory instruction then
        $a \leftarrow I(addr)$;
        if $I$ is a def then
            update last writer of $a$ in $M$ ;
        else if use then
            $w \leftarrow$ find last-writer of $a$ from $M$;
            if $u \rightarrow w$ edge $e \notin$ CDFG then
                add $e$ in $E_D$;
            foreach $i : it_a[i] \neq it_0[i]$ do
                $loop\_carried_e[i] \leftarrow true$;
                $set_e[i] \leftarrow set_e[i] \cup \{a\}$;
            $it_a \leftarrow it_0$;
    else if $I$ is an allocation instruction then
        $a \leftarrow I(addr)$;
        if $I$ is local then
            add $\{I(id), [a, a + I(size)]\}$ in $D_f$;
        else if $i$ is global $\vee$ alloc then
            add $\{I(id), [a, a + I(size)]\}$ in $GD_f$;
```

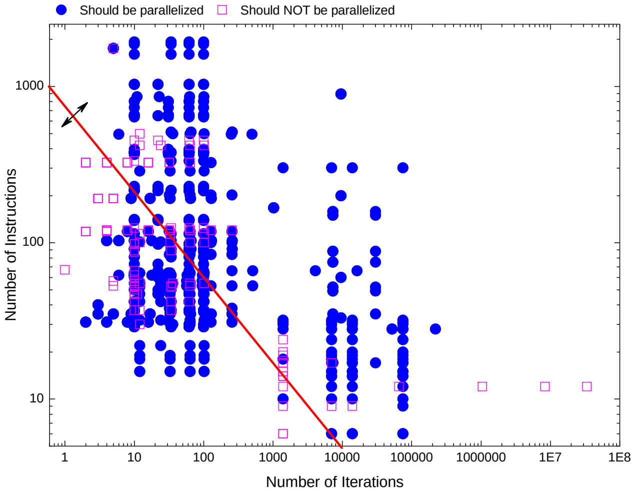


Fig. 5. This diagrams shows the optimal classification (sequential/parallel execution) of all parallel candidates considered in our experiments for the Intel Xeon machine. Linear models and static features such as the iteration count and size of the loop body in terms of IR statements are not suitable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

such a loop as potentially parallelizable, but present it to the user for final approval if parallelization is predicted to be profitable.

4.3. Parallelism Detection

Based on the dependence information provided by the constructed CDFG, we are able to efficiently detect parallelism. There are several particular issues need to be addressed in the parallelism detection stage, including detection of *parallel loops*, *privatizable variables* and *reduction operations* using standard analysis.

Parallel Loops. Parallel loops are discovered by traversing the loop nest trees of the CDFG in a top-down fashion. For each loop all the data-dependence edges that flow between nodes of the specific loop are processed. Each dependence edge is annotated with a bit vector that specifies which loop level a *loop-carried* dependence corresponds to. Based on this information and the level of the current loop, we can determine whether this particular edge prohibits parallelization or otherwise we proceed with the next edge.

Privatizable Variables. We maintain a complete list of true-, anti- and output-dependence 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 [Kennedy and Allen 2002]. 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 [Pottenger 1995]. We validate statically detected reduction candidates using profiling information.

4.4. Parallel Code Generation

Once the parallelism has been detected, our tool generates parallel code using OpenMP annotations. 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 (i.e. `omp parallel for (reduction)`).

Privatization Variables. After determining that a variable requires privatization and it is permitted to be privatized, we add a special `private` OpenMP clause with the list of these variables at the end of the parallel loop directive (clause `private(var1, var2, ...)`). The discovery of `firstprivate` and `lastprivate` variables is currently not supported by our analysis framework. In the case of a global variable, however, there are two cases that require different handling. If there is no function called within the loop body that accesses this variable we can still use the `private` clause. Otherwise, we add a `threadprivate` construct after its definition to make this variable globally private. If the thread-private global variable is not privatisable in all the parallel loops that this variable is accessed it should be renamed in this loop and any functions that are accessed within the loop body.

Reduction Operations. 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.

Limitations of Code Generation. In this article, our approach to code generation is relatively simple and, essentially, relies on OpenMP code annotations alongside minor code transformations. This framework does not 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. Our recent work extends this framework to generate code for task and pipeline parallelism, demonstrating the effectiveness of profiling analysis in exploiting automatic parallelization [Tournavitis and Franke 2010].

5. MACHINE LEARNING BASED MAPPING

The mapping stage decides if a parallel loop candidate may be *profitably* parallelized and, if so, selects a scheduling policy from the four options offered by OpenMP: *CYCLIC* (i.e. OpenMP clause `schedule(static,1)`), *DYNAMIC*, *GUIDED*, and *STATIC*. As the example in figure 2 demonstrates, this is a non-trivial task and the best 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.

Building and using a such model follows the well-known 3-step process [Bishop 2007]: (i) selecting a model and features (ii) training a predictive model (iii) deploying the model.

5.1. Model and Features

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 a slow-down due to an excessive synchronization overhead. Traditional parallelizing compilers such as SUIF-1 [Hall et al. 1996] employ simple heuristics

(1) Baseline SVM for classification
   (a) Training data:
   $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) | \boldsymbol{x}_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^{n}$
   (b) Maximum-margin hyperplane formulation:
   $C(\boldsymbol{w} \cdot \boldsymbol{x}_i - b) \geq 1$, 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.

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

based on the iteration count and the number of operations in the loop body to decide whether or not a particular parallel loop candidate should be executed in parallel.

Empirical data, as shown in figure 5, suggests that such a naïve scheme is likely to fail and that misclassification occurs frequently. Figure 5 plots loop bodies as a function of the number of iterations of a loop and the number of instructions it contains. 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 article we propose a *predictive modeling* approach based on machine-learning classification. In particular, we use *Support Vector Machines* (SVM) [Boser et al. 1992] 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 [Boser et al. 1992]. The details of our SVM classifier are provided in figure 6.

Program Features. The SVM model predicts which loops can be profitably parallelized as a function of the essential characteristics or *program features* of the loops. We extract such *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 I. The *static features* are derived from the internal code representation. Essentially, these features characterize the amount of work carried out in the parallel loop similar to e.g. [Ziegler and Hall 2005]. The *dynamic features* capture the dynamic data access and control flow patterns of the *sequential* program and are obtained from the same profiling execution that has been used for parallelism detection.

We select features that are important for performance from a programmer's point of view and shown to be crucial for performance in other work. For example, the number of instructions and the loop iteration count implicitly capture the amount of work to be performed within the loop body. Similarly, the load and store count and the cache miss rate strongly correlate to the communication and we thus include them in the feature set. The branch miss rate is important for evaluating the profitability on the Cell processor. Finally, as the cost of synchronization can be expensive on some platforms, we also include this feature in table I.

Table I. Features Used.

Static Features	Dynamic Features
IR Instruction Count	L1/L2 DCache Miss Rate
IR Load/Store Count	Instruction Count
IR Branch Count	Branch Miss Prediction Rate
Loop Iteration Count	Synchronization Count

5.2. Training

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.

The training task here is to construct hyper-planes on the program feature space so that the trained SVM classifier can accurately predict mappings for *unseen* programs. Before constructing any separating hyper-planes, we need to set up the SVM model parameters. There are two parameters to be determined for a SVM model with the RBF kernel: C and γ (see figure 6). It is not known beforehand what values of these parameters are best for the problem; hence, some kind of parameter search must be performed. We perform the parameter search as follows. Initially, we generate many pairs of the two parameters, $(C; \gamma)$. For each pair of the parameters, we first randomly split the *training* data into two different sets – one is used for building hyper-planes with a particular parameter pair and the other is used for evaluating the effectiveness of the parameter pair; then, we train a SVM model using the hyper-plane building data set, evaluate performance of the trained model using the parameter evaluation set, and record the prediction accuracy. For each pair of model parameters, we repeat this procedure (i.e. randomly partitioning the whole training data into two sets) several times and calculate the average prediction accuracy of the SVM models. As a result, we pick the pair of parameters that gives the best average accuracy and use it to train a final model by using the *whole* training data set. Note that during the training process, the training algorithm only performs on the training data set.

The process of parameter search is exemplified in the contour map shown in figure 7. In this figure a contour line is a curve that joints different value pairs of the two model parameters, $(C; \gamma)$, which are shown as x and y axis. Different curves represent different prediction accuracy for SVM models with specific parameter settings. As can be seen from this figure there are actually many parameter settings that give high prediction accuracy (i.e. 93% of accuracy in figure 7). In other words the accuracy of the model is not very sensitive to particular parameter values.

5.3. Deployment

Once we have determined the right model parameters and built the model from the training data, we can now use the model to predict the mapping of a *new, unseen* program. For a new, previously unseen application with parallel annotations the following steps are carried out:

(1) *Feature extraction*. This involves collecting the features shown in table I 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 exe-

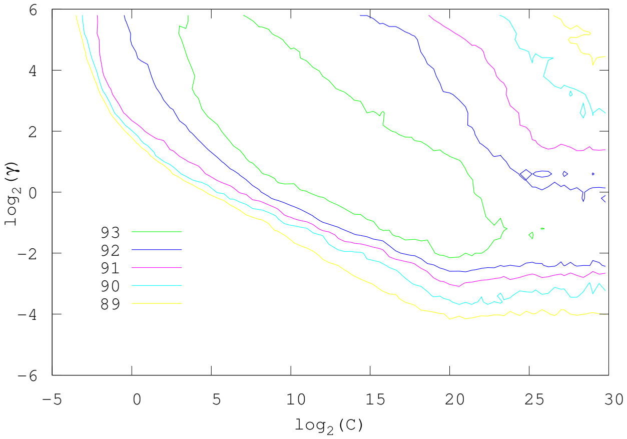


Fig. 7. Finding suitable SVM model parameters using grid search. In this contour map each curve connects points of a pair of the two parameters that gives a particular prediction accuracy. The parameter setting that leads to the best prediction accuracy is chosen as the final model parameters.

cution 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 be 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.

6. SAFETY AND SCALABILITY ISSUES

This section provides detailed discussions about the safety and scalability issues of our profiling-driven parallelism detection approach.

6.1. 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 provides a straightforward way to select representative workloads (in terms of data-dependency) 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 CPU2000 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 vari-

Table II. Hardware and software configurations.

Intel Xeon Server	
Hardware	Dual Socket, 2× Intel Xeon X5450 @ 3.00GHz 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 -O2 -xT -axT -ipo
Cell Blade Server	
Hardware	Dual Socket, 2× IBM Cell processors @ 3.20GhZ 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

ables 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.

Alternatively, thread-level-speculation provides a rich set of techniques that detect and recover from incorrect speculation during runtime execution [Rauchwerger and Padua 1995; Dou and Cintra 2004; Johnson et al. 2012]. These techniques can be used to ease the burden of user verification, which is the future work of our framework.

6.2. 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. Other dynamic trace analysis and compression techniques, such as dynamic tree compression [Ding and Zhong 2003] and ZDDs [Price and Vachharajani 2010], can be used to further reduce the size of the trace and to improve the analysis efficiency, which is the future work.

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.

7. EXPERIMENTAL METHODOLOGY

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

7.1. 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 II.

Table III. Benchmark applications and data sets.

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

7.2. Benchmarks

For our evaluation we have selected benchmarks – NAS Parallel Benchmarks (NPB) and SPEC CPU2000 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 [omm ; Aslot et al. 2001]. For the NPB benchmarks, we used four input classes, i.e. S, W, A, and B, when possible. Class S is the smallest input, class B is the largest input for a single machine and classes W and A are medium-sized inputs. Due to the memory constraint of the CELL processor, it is impossible to compile some of the programs with classes A and B and we excluded those input sets. Benchmark *BT* is also excluded on the Cell platform because the compiler fails to compile it.

More specifically, we have used the NAS NPB sequential v.2.3 and NPB OpenMP v.2.3 codes [Bailey et al. 1991] 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.

Each program has been executed using multiple different input data sets (shown in table III), however, for parallelism detection and mapping we have only used the *smallest* of the available data sets[2]. 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.

7.3. Methodology

We have evaluated three different parallelization approaches, which are *manual*, *auto-parallelization* using the Intel ICC compiler (just for the Xeon 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 single-source XLC compilers for the Intel Xeon and IBM Cell platforms, respectively. We used the compiler flags that Intel used for its SPEC performance submission. This gives the best averaged performance for the compiler version we used. For the CELL platform, the IBM single-source compiler automatically generates parallel threads to utilize the SPE accelerators with the SIMD auto-vectorization support. It also comes with an associated runtime that exploits a software cache to hide the communication latency between the host PPE and SPEs. Each experiment was repeated 10 times and the median execution time was recorded. As we found in the experiments, there is small variances in each execution (less than 5%).

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

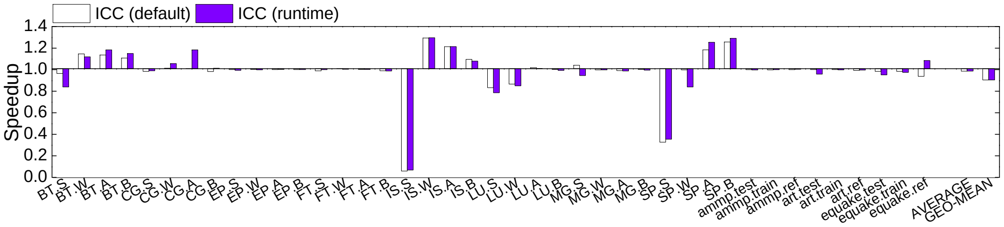


Fig. 8. Speedup over sequential code achieved by the Intel ICC auto-parallelizing compiler.

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 Power Processor Element (PPE) rather than a single Synergistic Processing Element (SPE). In all cases the sequential performance of the PPE exceeds that of a single SPE, ensuring we report improvements over the *strongest* baseline available. The average performance is presented with both arithmetic and geometric means.

8. EXPERIMENTAL EVALUATION

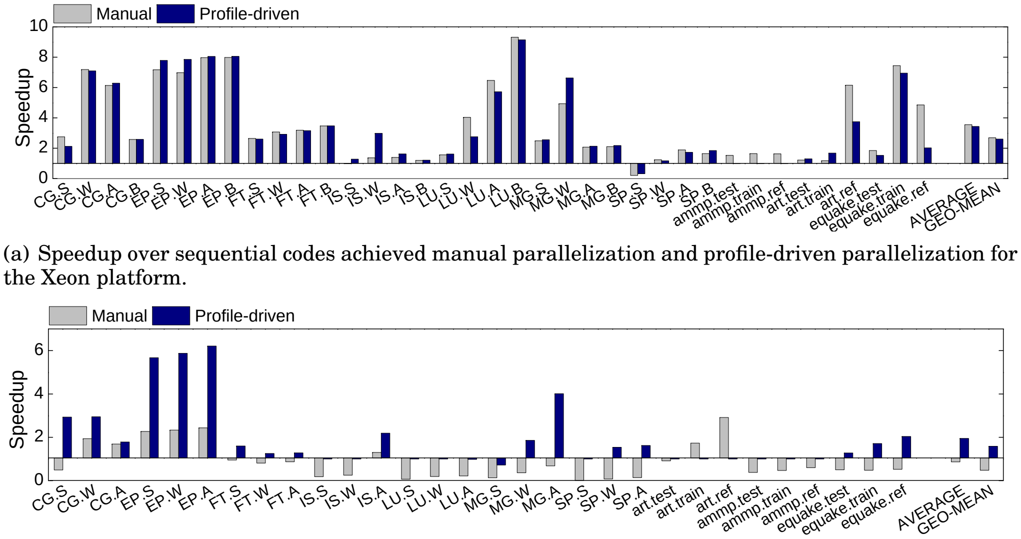
In this section we present and discuss our experimental results on the Xeon and Cell platforms. Firstly, we present the overall results of our framework. Next, we show that our profiling-driven parallelism detection approach is very efficient in discovering parallelism. Finally, we compare our machine-learning based mapping model to fixed heuristics which use the same profiling information.

8.1. Overall Results

Intel Xeon. Before we evaluate our approach, it is useful to examine the performance of an existing auto-parallelizing compiler currently available on the Intel Xeon.

ICC. Figure 8 shows the performance achieved by the Intel ICC with two different parallelization settings: default and runtime. With the default setting the ICC compiler uses a default profitability threshold to decide if a loop should be parallelized while with the runtime setting, the compiler uses runtime checking to decide the probability threshold. Overall ICC 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 largest data set sizes, respectively. ICC gains a modest speedup only for the larger data sets of the *IS* and *SP* benchmarks. Though the ICC (runtime) approach achieves better performance than the default scheme for some applications, it results in significant slowdown of the *BT.S* and *SP.S* benchmarks due to the overhead of runtime checking. The reason for the disappointing performance of the Intel ICC compiler is that it typically parallelizes at the inner-most loop level where significant fork/join overhead negates the potential benefit from parallelization. On average, both parallelization settings of ICC result in 2% of slowdown over the sequential code.

Manual. Figure 9(a) summarizes the performance of our scheme and manually parallelized OpenMP programs. The manually parallelized OpenMP programs achieve an arithmetic mean-speedup of 3.5 (a geometric mean speedup of 2.69) 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



(a) Speedup over sequential codes achieved 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.

Fig. 9. Speedups due to different parallelization schemes.

caching [Grant and Afsahi 2007]. 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 [omm ; Grant and Afsahi 2007].

Profile-driven. For most NAS benchmarks our profile-driven parallelization achieves performance levels close to those of the manually parallelized versions. In the manually parallelized version, parallel loops were put into big parallel sections, leading to better performance on small inputs due to the reduction of thread spawning overhead. However, this is not a significant advantage for medium and large inputs where the parallel execution dominates the whole program execution time. Our profile-driven approach outperforms the manual version on some benchmarks (*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 [Aslot et al. 2001]. 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. As shown in figure 10, 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 on the Xeon platform. 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 parallel version is 6 times faster than the sequential SPEC CPU2000 version, of which more than 50% is due to sequential code optimizations. We

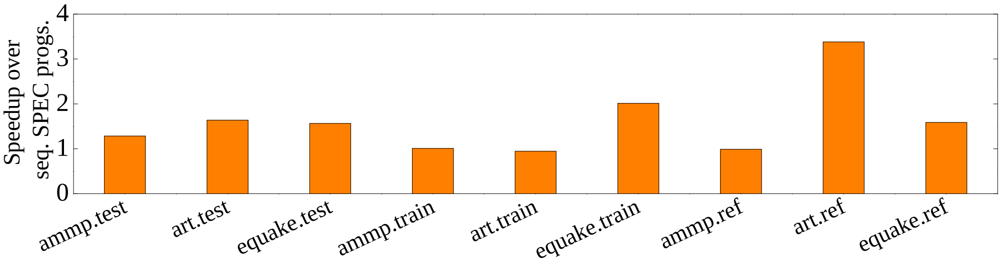


Fig. 10. The sequential performance of the SPEC OMP2001 benchmarks compared to the original SPEC CPU2000 benchmarks on the Xeon platform. The SPEC OMP benchmarks have better sequential performance than the original SPEC CPU2000 counterparts due to the reconstruction of codes.

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 arithmetic speedup of 3.45 (a geometric mean speedup of 2.6) across all benchmarks.

IBM Cell. Figure 9(b) shows the performance resulting from manual and profile-driven parallelization for the dual-Cell platform. Benchmark *BT* is not included in this figure because the IBM CELL compiler fails to compile it. Unlike the Intel platform, the Cell does not deliver 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 has not considered the communication costs for a distributed memory machine. By contrast, our mapping scheme is able to exclude most of the non-profitable loops for parallelization, leading to better performance. 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 an arithmetic mean-speedup of 2.0 (which translates to a geometric mean-speedup of 1.60). 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.

8.2. 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,

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

	Profile-driven			ICC	Manual
App.	#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%)

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 IV 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. Clearly, this result does not prove that dynamic analysis is always correct. However, it indicates that profile-based dependence analysis may be more accurate than generally considered, even for profiles generated from small data set. This encouraging result will need further validation on more complex programs before we can draw any final conclusions.

Column 3 in table IV 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 8.3).

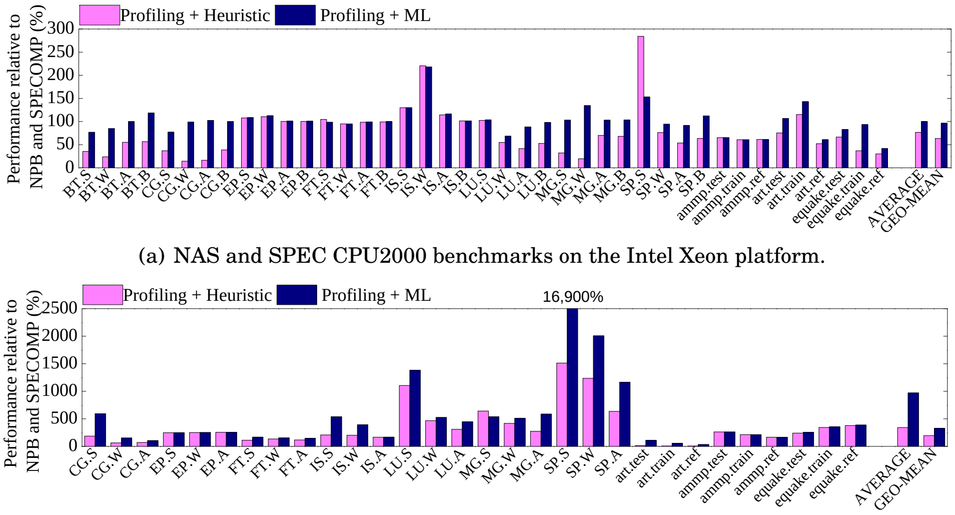
The final column in table IV eventually shows the number of loops parallelized in the hand-coded applications. As before, the percentage of sequential coverage is shown in parentheses. Far fewer loops than theoretically possible are actually parallelized because the programmer have obviously decided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

8.3. Parallelism Mapping

In this section we evaluate the impact of machine-learning driven mapping on performance. In order to isolate its effect we examine three mapping schemes: manual; a heuristic; and machine-learning based predictive model; across the two platforms[3]. The heuristic is typical profitable model that is similar to the one used in SUIF [Hall

[3] The results of ICC are not shown in these experiments, because ICC gives little speedup over the sequential version of the code (see section 8.1).



(a) NAS and SPEC CPU2000 benchmarks on the Intel Xeon platform.

(b) NAS and SPEC CPU2000 benchmarks on the IBM Cell platform.

Fig. 11. Impact of different mapping approaches (100% = manually parallelized OpenMP code).

et al. 1996] and Open64 [Open64 2013] parallelizing compilers. It uses the profiling information to calculate the execution time per loop iteration and decides the profitability of loops based on a certain threshold. In the experiments, we have tried different thresholds and used the one that gives the best average performance on a particular platform.

Intel Xeon. Figure 11(a) compares 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 76% of the performance of the hand-parallelized code. This translates into a geometric mean speedup of 1.7 rather than 2.6 for the benchmarks. The main reason for this performance loss is that the fixed heuristic is unable to accurately determine whether a loop should be parallelized or not. There are two cases, *FT.S* and *SP.S*, that the fixed heuristic achieves slightly better performance than our machine learning model. This is because our machine learning model is over-optimistic for the two applications with the smallest data sets by parallelizing several unprofitable loops. This can be improved by adding more training examples.

IBM Cell. The diagram in figure 11(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. The combination of profiling and machine-learning dramatically outperforms the fixed heuristic counterpart by more than a factor of 2. This, on average, results in a

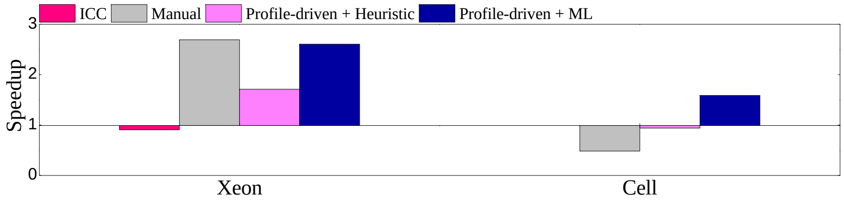


Fig. 12. Geometric mean speedups on the two platforms.

geometric mean speedup of 3.31 over the hand-parallelized OpenMP programs across all data sets. There is only one case (i.e., *MG.S* that the fixed heuristic outperforms the machine learning model which is over-optimistic in evaluating the profitability of some loops for this program. This can be easily improved by using more training examples when training the machine learning model.

Summary. Figure 12 shows the average speedups over the sequential code for each approach on the two platforms. 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. It achieves a geometric mean speedup of 2.60 over the sequential code, which is very close to the 2.69x speedup achieved by the manual parallelization code. 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 Cell platform where accurate mapping decisions are key enablers to high performance. Existing ("generic") manually parallelized OpenMP code fails to deliver any reasonable performance, which actually results in 50% of slowdown in performance. Static heuristics, even with carefully chosen profitability threshold, are unable to match the performance of our machine-learning based scheme.

8.4. Development Cost

Profiling. The time spent on profiling and analysis depends on the program to be parallelized and the input data set used for profiling. In our case we can profile and analyze all the programs in less than an hour with up to 100x slowdown over the sequential execution. The potential performance improvement could be a strong incentive for developers to adopt this automatic approach. There are also techniques, such as [Kim et al. 2010], can further reduce the profiling and analysis overhead.

Training. The task of generating and collecting training data was done within a day for both platforms. More specifically, running programs and collecting training data took 16 hours, and training the machine learning model took about 10 minutes. The process of collecting data and training model is a one off cost incurred by our framework. Furthermore, generating and collecting data is a completely automatic process and is performed off-line. Therefore, it requires far less effort than constructing a heuristic for each platform by hand.

Prediction. It takes a few microseconds to evaluate the model and make a prediction. Thus the overhead of prediction is negligible which is included the experimental results.

9. RELATED WORK

There is a substantial body of literature in automatic parallelization and related areas. Here we briefly survey some of the large body of related work and discuss recent development since [Tournavitis et al. 2009] was published.

Parallel Programming Languages. Many approaches have been proposed for new programming languages to enable easier exploiting of parallelism [Frigo et al. 1998; Gordon et al. 2002; Saraswat et al. 2007]. While such languages are critical in the long term these approaches do not alleviate the problems of porting legacy code.

Automatic Data-Parallelization. Automatic parallelism extraction has been achieved on restricted DOALL and DOACROSS loops [Kuck et al. 1981; Kennedy and Allen 2002; Burke and Cytron 1986; Padua et al. 1993; Lim and Lam 1997]. Unfortunately, many parallelization opportunities were ignored due to the lack of information at the source code level. The main problem is that for many programs, it is not compile-time decidable if there exists a dependence between two references. To guarantee safety, an overly conservative approach has to be employed limiting parallelization performance.

Dynamic Parallelization. To overcome the limits of static analysis there has been considerable work in runtime analysis and parallelization Dynamic dependence analysis [Peterson and Padua 1993; Rauchwerger et al. 1995; Chen and Olukotun 2003] and hybrid data dependence analysis [Rus et al. 2003] make use of dynamic dependence information, delaying much of the parallelization work to the runtime of the program. Such approaches record memory access patterns and execute in parallel if there is no conflict. This however comes at considerable runtime cost. In contrast, we employ a separate profiling stage and incorporate the dynamic information in the usual compiler based parallelization without causing any runtime overhead. Rus et al. [Rus et al. 2007] applied sensitivity analysis to automatic parallelize programs whose behaviors may be sensitive to input data sets. Sensitive analysis uses runtime information (such as loop bounds) to valid parallelization. In contrast to their approach, our profiling-driven approach discovers more parallel opportunities and selecting parallel candidates and scheduling policies across multiple architectures.

Speculative Parallelization. In a related area to dynamic parallelization, there are other approaches to exploiting parallelism in a speculatively execution manner [Rauchwerger and Padua 1995], but these approaches usually require hardware support for efficient execution. Prabhu and Olukotun [Prabhu and Olukotun 2005], and Bridges et al. [Bridges et al. 2007] have manually parallelized the SPECINT-2000 benchmarks with thread level speculation. Their approaches rely upon the programmer to discover parallelism as well as runtime support for parallel execution. Decoupled Software Pipelining (`DSWP`) [Rangan et al. 2004; Ottoni et al. 2005] extracts thread parallelism where the parallel threads communicate in a pipelined manner using software or hardware queues. `DSWP` requires fine-grain communication between parallel threads, making it infeasible for heterogeneous multi-cores where the communication cost is expensive. Privateer is a compiler framework [Vachharajani et al. 2007] that targets parallelization of DOALL loops by providing supports of speculative privatization and reductions. Profiling information is also used to identify loop candidates that can benefit from speculative execution [Du et al. 2004; ct0 2004; Wu et al. 2008]. None of these approaches addresses the problem of mapping parallelism across different platforms.

Interactive Parallelization. Interactive parallelization tools [Brandes et al. 1997; Ishihara et al. 2006] provide a way to actively involve the programmer in the detection and mapping of application parallelism. For example, SUIF Explorer [Liao et al. 1999] 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* [Ding et al. 2007] system allows the programmer to specify intended parallelism. In [Thies et al. 2007], programmers mark potential parallel regions of the program then the tool uses dynamic profiling information to find a good mapping of parallel candidates. All these frameworks require the programmer to mark parallel regions instead of discovering parallelism automatically.

Parallelism Mapping. Prior research in parallelism mapping has mainly focused on building heuristics and analytical models [Ramanujam and Sadayappan 1989], runtime adaptation [Corbalán et al. 2000] approaches, and on mapping or migrating tasks on a specific platform. In this article, we aim to develop a compiler-based, automatic, and portable approach that can adapt to multi-core hardware.

Adaptive Compilation. Pouchet et al. combine iterative compilation and model driven search in a single framework [Pouchet et al. 2010]. They first formulate a set of loop transformations, including loop tiling and vectorization, in a static, polyhedral modeling based space. For a given loop, their framework empirically searches over valid transformations in the space to find the best transformation sequence for the target architecture. They show that a single program version does not perform equally well on different platforms and thus an adaptive optimization scheme is needed. 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. Our auto-parallelization framework is built on top of our early work [Wang and O'Boyle 2009] where the input program of [Wang and O'Boyle 2009] has to be manually parallelized by programmers.

Recent Research Development. Since the early work of this article was published [Tournavitis et al. 2009], a significant volume of research work has adopted similar techniques but develops into different directions. Some of the work use profiling analysis to develop interactive parallelization frameworks. A such framework runs the sequential program to uncover possibly parallel regions and to help the user select the most profitable parallel regions with speedup estimation [Garcia et al. 2011]. Other work applies dynamic analysis to either exploit different types of parallelism or target different application domains. The Parallax framework [Vandierendonck et al. 2010] exploits pipeline parallelism using user annotations. To help the programmer annotates the code, it uses profiling information to suggest that where an explicit annotation is required in the source code in order to safely parallelize the sequential program.Like our approach, Paralax uses static and profiling information to construct a program dependence graph to perform dependence analysis.Other examples include the McFLAT framework which uses profiling runs to decide what transformations to apply to Matlab programs [Aslam and Hendren 2010]. In addition to parallelism detection, techniques are proposed to reduce the profiling overhead. For example, SD3 [Kim et al. 2010] uses parallel threads to perform the profiling analysis so as to reduce the runtime overhead. It also performs dependence analysis on the compressed trace to reduce memory footprint. Experimental results show that SD3 can greatly reduce the runtime and memory overhead when profiling the SPEC 2006 benchmark suite. Dave and Eigenmann consider the problem of tuning OpenMP programs for multicores [Dave and Eigenmann 2009]. Their approach uses iterative compilation and executions to decide the granularity of parallelism, which can deliver close and even better performance than hand-parallelized code. Recently, Wang and O'Boyle propose a machine learning based approach to partitioning StreamIt programs onto shared memory multi-cores [Wang and O'Boyle 2010; Wang and O'boyle 2013]. This approach predicts the ideal structure of the program graph and then uses random search to generate a program partition that is closed to the predicted structure. Predictive modelling has also been used to optimize parallel programs in various settings [Grewe et al. 2011; 2013; Grewe et al.] In industrial terms, Profile-driven analysis has recently been adopted by companies like Vector Fabrics [Fabrics 2013] to help programmers to generate optimised parallel code for multi-cores.

10. CONCLUSION AND FUTURE WORK

In this article we have developed a platform-agnostic, profiling-based parallelism detection method that enhances static data dependence analysis with dynamic information, resulting in larger amounts of parallelism uncovered from sequential programs. We have also shown that a close interaction with an adaptive mapping scheme can be successful. Our mapping scheme is automatically built from training data, which is portable across different 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 profile-driven approach can discover more parallelization opportunities and the machine learning based mapping scheme is more portable than existing static mapping strategies. Our holistic approach is able to achieve 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

NAS parallel benchmarks, OpenMP C version. `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

2004. In *CC '04*.

ASANOVIC, K., BODIK, R., DEMMEL, J., KEAVENY, T., KEUTZER, K., KUBIATOWICZ, J., MORGAN, N., PATTERSON, D., SEN, K., WAWRZYNEK, J., WESSEL, D., AND YELICK, K. 2009. A view of the parallel computing landscape. *Communications of ACM 52,* 10, 56–67.

ASLAM, A. AND HENDREN, L. 2010. Mcflat: a profile-based framework for matlab loop analysis and transformations. In *LCPC'10*. 1–15.

ASLOT, V., DOMEIKA, M. J., EIGENMANN, R., GAERTNER, G., JONES, W. B., AND PARADY, B. 2001. Specomp: A new benchmark suite for measuring parallel computer performance. In *WOMPAT '01*. 1–10.

BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. K. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Supercomputing '91*. 158–165.

BISHOP, C. M. 2007. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer.

BOSER, B. E., GUYON, I. M., AND VAPNIK, V. N. 1992. A training algorithm for optimal margin classifiers. In *COLT '92*. 144–152.

BRANDES, T., CHAUMETTE, S., COUNILH, M. C., ROMAN, J., DARTE, A., DESPREZ, F., AND MIGNOT, J. C. 1997. Hpfit: a set of integrated tools for the parallelization of applications using high performance fortran. part i: Hpfit and the transtool environment. *Parallel Computing 23*, 71–87.

BRIDGES, M., VACHHARAJANI, N., ZHANG, Y., JABLIN, T., AND AUGUST, D. 2007. Revisiting the sequential programming model for multi-core. In *MICRO 40*. 69–84.

BURKE, M. AND CYTRON, R. 1986. Interprocedural dependence analysis and parallelization. In *CC '86*. 162–175.

CHEN, M. K. AND OLUKOTUN, K. 2003. The jrpm system for dynamically parallelizing java programs. In *ISCA '03*. 434–446.

CORBALÁN, J., MARTORELL, X., AND LABARTA, J. 2000. Performance-driven processor allocation. In *OSDI'00*. 5–17.

COSY. 2009. CoSy compiler development system. `http://www.ace.nl/compiler/`.

DAVE, C. AND EIGENMANN, R. 2009. Automatically tuning parallel and parallelized programs. In *LCPC'09*. 126–139.

DING, C., SHEN, X., KELSEY, K., TICE, C., HUANG, R., AND ZHANG, C. 2007. Software behavior oriented parallelization. In *PLDI '07*. 223–234.

DING, C. AND ZHONG, Y. 2003. Predicting whole-program locality through reuse distance analysis. In *PLDI '03*.

DOU, J. AND CINTRA, M. 2004. Compiler estimation of load imbalance overhead in speculative parallelization. In *PACT '04*. 203–214.

DU, Z.-H., LIM, C.-C., LI, X.-F., YANG, C., ZHAO, Q., AND NGAI, T.-F. 2004. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI '04*. 71–81.

FABRICS, V. 2013. http://www.vectorfabrics.com/.

FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. 1998. The implementation of the cilk-5 multithreaded language. In *PLDI '98*. 212–223.

GARCIA, S., JEON, D., LOUIE, C. M., AND TAYLOR, M. B. 2011. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI '11*. 458–469.

GORDON, M. I. 2010. Ph.d. thesis. Ph.D. thesis, Massachusetts Institute of Technology.

GORDON, M. I., THIES, W., KARCZMAREK, M., LIN, J., MELI, A. S., LAMB, A. A., LEGER, C., WONG, J., HOFFMANN, H., MAZE, D., AND AMARASINGHE, S. 2002. A stream compiler for communication-exposed architectures. In *ASPLOS-X*. 291–303.

GRANT, R. E. AND AFSAHI, A. 2007. A comprehensive analysis of openmp applications on dual-core Intel Xeon SMPs. In *IPDPS '07*. 1–8.

GREWE, D., WANG, Z., AND O?BOYLE, M. F. Opencl task partitioning in the presence of gpu contention. In *LCPC '13*.

GREWE, D., WANG, Z., AND O'BOYLE, M. F. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *CGO '13*.

GREWE, D., WANG, Z., AND O'BOYLE, M. F. P. 2011. A workload-aware mapping approach for data-parallel programs. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. HiPEAC '11. 117–126.

HALL, M., ANDERSON, J., AMARASINGHE, S., MURPHY, B., LIAO, S.-W., AND BU, E. 1996. Maximizing multiprocessor performance with the SUIF compiler. *Computer 29,* 12, 84–89.

HUSBANDS, P., IANCU, C., AND YELICK, K. 2003. A performance analysis of the berkeley upc compiler. In *ICS '03*. 63–73.

IRIGOIN, F., JOUVELOT, P., AND TRIOLET, R. 1991. Semantical interprocedural parallelization: an overview of the pips project. In *ICS '91*. 244–251.

ISHIHARA, M., HONDA, H., AND SATO, M. 2006. Development and implementation of an interactive parallelization assistance tool for openmp: ipat/omp. *IEICE Transactions on Information and Systems E89-D*, 399–407.

JOHNSON, NICK P.AND KIM, H., PRABHU, P., ZAKS, A., AND AUGUST, D. I. 2012. Speculative separation for privatization and reductions. In *PLDI '12*.

KENNEDY, K. AND ALLEN, J. R. 2002. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc.

KENNEDY, K., MCKINLEY, K. S., AND TSENG, C. W. 1991. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems 2*.

KIM, M., KIM, H., AND LUK, C.-K. 2010. Sd3: A scalable approach to dynamic data-dependence profiling. In *MICRO '43*. 535–546.

KUCK, D. J., KUHN, R. H., PADUA, D. A., LEASURE, B., AND WOLFE, M. 1981. Dependence graphs and compiler optimizations. In *POPL '81*. 207–218.

KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., AND CHEW, L. P. 2007. Optimistic parallelism requires abstractions. In *PLDI '07*. 211–222.

LAMPORT, L. 1974. The parallel execution of do loops. *Communications of ACM 17*, 83–93.

LIAO, S.-W., DIWAN, A., BOSCH, JR., R. P., GHULOUM, A., AND LAM, M. S. 1999. SUIF explorer: an interactive and interprocedural parallelizer. In *PPoPP '99*. 37–48.

LIM, A. W. AND LAM, M. S. 1997. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97*. 201–214.

OPEN64. 2013. http://www.open64.net.

OTTONI, G., RANGAN, R., STOLER, A., AND AUGUST, D. I. 2005. Automatic thread extraction with decoupled software pipelining. In *MICRO 38*. 105–118.

PADUA, D. A., EIGENMANN, R., HOEFLINGER, J., PETERSEN, P., TU, P., WEATHERFORD, S., AND FAIGIN, K. 1993. Polaris: A new-generation parallelizing compiler for mpps. Tech. rep., University of Illinois at Urbana-Champaign.

PETERSON, P. AND PADUA, D. A. 1993. Dynamic dependence analysis: A novel method for data depndence evaluation. In *LCPC '93*. 64–81.

POTTENGER, W. M. 1995. Induction variable substitution and reduction recognition in the polaris parallelizing compiler. Tech. rep., University of Illinois at Urbana-Champaign.

POUCHET, L.-N., BONDHUGULA, U., BASTOUL, C., COHEN, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC '10*.

PRABHU, M. K. AND OLUKOTUN, K. 2005. Exposing speculative thread parallelism in SPEC2000. In *PPoPP '05*. 142–152.

PRICE, G. AND VACHHARAJANI, M. 2010. Large program trace analysis and compression with zdds. In *CGO '10*.

RAMANUJAM, J. AND SADAYAPPAN, P. 1989. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Supercomputing '89*.

RANGAN, R., VACHHARAJANI, N., VACHHARAJANI, M., AND AUGUST, D. I. 2004. Decoupled software pipelining with the synchronization array. In *PACT '04*. 177–188.

RAUCHWERGER, L., AMATO, N. M., AND PADUA, D. A. 1995. Run-time methods for parallelizing partially parallel loops. In *ICS '95*. 137–146.

RAUCHWERGER, L., ARZU, F., AND OUCHI, K. 1998. Standard templates adaptive parallel library (stapl). In *LCPC '98*. 402–409.

RAUCHWERGER, L. AND PADUA, D. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI '95*. 218–232.

RUL, S., VANDIERENDONCK, H., AND BOSSCHERE, K. D. 2008. A dynamic analysis tool for finding coarse-grain parallelism. In *HiPEAC Industrial Workshop*.

RUS, S., PENNINGS, M., AND RAUCHWERGER, L. 2007. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS '07*. 263–273.

RUS, S., RAUCHWERGER, L., AND HOEFLINGER, J. 2003. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming 31*, 251–283.

SARASWAT, V. A., SARKAR, V., AND VON PRAUN, C. 2007. X10: concurrent programming for modern architectures. In *PPoPP '07*. 271–271.

THIES, W., CHANDRASEKHAR, V., AND AMARASINGHE, S. 2007. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO 40*. 356–369.

TOURNAVITIS, G. AND FRANKE, B. 2010. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *PACT '10*. 377–388.

TOURNAVITIS, G., WANG, Z., FRANKE, B., AND O'BOYLE, M. F. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09*. 177–187.

VACHHARAJANI, N., RANGAN, R., RAMAN, E., BRIDGES, M. J., OTTONI, G., AND AUGUST, D. I. 2007. Speculative decoupled software pipelining. In *PACT '07*. 49–59.

VANDIERENDONCK, H., RUL, S., AND DE BOSSCHERE, K. 2010. The paralax infrastructure: automatic parallelization with a helping hand. In *PACT '10*. 389–400.

WANG, Z. AND O'BOYLE, M. F. 2009. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09*.

WANG, Z. AND O'BOYLE, M. F. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT'10*.

WANG, Z. AND O'BOYLE, M. F. P. 2013. Using machine learning to partition streaming programs. *ACM ACM Transactions on Architecture and Code Optimization 10,* 3, 1–25.

WU, P., KEJARIWAL, A., AND CAŞCAVAL, C. 2008. Compiler-driven dependence profiling to guide program parallelization. In *LCPC '08*. 232–248.

ZIEGLER, H. AND HALL, M. 2005. Evaluating heuristics in automatically mapping multi-loop applications to FPGAs. In *FPGA '05*. 184–195.